Lift: An Educational Interactive Stochastic Ray Tracing Framework with AI-Accelerated Denoiser

Gonçalo Soares
Instituto Superior Técnico/Inesc-ID
Universidade de Lisboa
Lisboa, Portugal
goncalofdsoares@tecnico.ulisboa.pt

João Madeiras Pereira
Instituto Superior Técnico/Inesc-ID
Universidade de Lisboa
Lisboa, Portugal
jap@inesc-id.pt

ABSTRACT

Real-time physically based rendering has long been looked at as the holy grail in Computer Graphics. With the introduction of Nvidia RTX-enabled GPUs family, light transport simulations under real-time constraint started to look like a reality. This paper presents Lift, an educational framework written in C++ that explores the RTX hardware pipeline by using the low-level Vulkan API and its Ray Tracing extension, recently made available by Khronos Group. Furthermore, to accomplish low variance rendered images, we integrated the AI-based denoiser available from the Nvidia´s OptiX framework. Lift's development arose primarily in the context of the graduate 3D Programming course taught at Instituto Superior Técnico and Master Theses focused on Real-Time Ray Tracing and provides the foundations for laboratory assignments and projects development. The platform aims to make easier students to learn and to develop, by programming the shaders of the RT pipeline, their physically-based rendering approaches and to compare them with the built-in progressive unidirectional and bidirectional path tracers. The GUI allows a user to specify camera settings and navigation speed, to select the input scene as well as the rendering method, to define the number of samples per pixel and the path length as well as to denoise the generated image either every frame or just the final frame. Statistics related with the timings, image resolution and total number of accumulated samples are provided too. Such platform will teach that nowadays physically-accurate images can be rendered in real-time under different lighting conditions and how well a denoiser can reconstruct images rendered with just one sample per pixel.

Keywords

Educational Ray Tracing framework, Nvidia RTX , Vulkan, Path Tracing, AI-accelerated Denoiser

1 INTRODUCTION

1.1 Motivation

For a long time, Real-Time Ray Tracing has been considered a far to reach dream where physically-based rendering would be calculated fast enough so that we would be able to interact with the scene and perceive the changes in real-time. Now that Nvidia RTX technology (Burgess, 2020) is available, the dream has become feasible more than ever before. Typical production renderers deal with offline image synthesis by using a large number of samples per pixel to render the best-looking image possible. Under time constraint, even the fastest ray tracers can only trace few rays per pixel at 1080p and 30Hz. While this number increases every few years, the trend is partially countered by the move towards higher resolution displays and higher refresh rates. It therefore seems likely that a realistic sampling budget for real-time applications will remain on the order of a low number of short paths per pixel which implies that the usage of Monte Carlo integration of indirect illumination leads to images containing very high levels of variance. To bridge this gap, recent research was able to design real-time reconstruction filters to remove the noise (denoise) from extremely low sample count images, thus allowing to synthesize noise-free images at real-time refresh rates (30Hz). These state-of-the-art denoisers often resort to machine learning methods like the works of (Chaitanya et al., 2017) and (Vicini et al., 2019). Over the years, multiple ray-tracing algorithms have been proposed to physically simulate the light transport and the interactions with materials of a scene under different lighting conditions. Now that it is possible to make these simulations at rates never seen before, it seems relevant to develop an educational framework that helps Master students to implement such algorithms by exploiting the RTX technology and by adding a denoising step, and then perform

a comparison between them to assess both the performance and the perceptual image quality.

1.2 Objectives

This paper describes the implementation of the Lift system which aims to provide an educational platform that makes easier students to learn and to program new light transport algorithms into a Vulkan RT-based working pipeline. Lift additionally provides a set of settings and statistics in order to facilitate the comparison of the students' programs with the built-in Monte Carlo algorithms, namely Path Tracing (PT) and Bidirectional Path Tracing (BDPT). The implementation of these two algorithms was leveraged on the graphics card RTX capabilities through the use of the Ray Tracing extension for the low-level Vulkan API (Daniel Koch Tobias Hector and Werness, 2020), recently made available by Khronos Group.

Furthermore, we integrated an AI-accelerated denoiser to be used together with both PT and BDPT algorithms.This denoiser is available in Optix (Parker et al., 2010), a programmable general-purpose ray tracing framework. The purpose was to create a platform that allows not only to achieve, in real-time, high-quality images with just one sample per pixel but also to check whether it is worth investing in more complex raytracing algorithms, like the BDPT algorithm, when compared to the unidirectional path tracer with denoiser.

Since OptiX's denoiser was our choice, we could use this API also to develop the internal RT working pipeline. However, one of the requisites of the 3D Programming Master course at Instituto Superior Técnico (IST) is to teach the Vulkan low level API and since its RTX extension was made available this year we decided to use Vulkan in the Lift framework. The development of the prototype was technically challenging because we had to build the Vulkan interoperability with the OptiX framework.

Summarizing, Lift is an educational framework that will help advanced students to learn and to tackle the following state-of-the-art questions:

- Can Ray Tracing algorithms render in real-time physically-accurate images, under different lighting conditions?
- How well a denoiser behaves in reconstructing images rendered with one sample per pixel?
- Is it worth investing on complex light transport algorithms over the simple path tracer with denoising?

In order to check if Lift was robust enough to address properly the above questions, we gathered data from multiple scenes rendered with both algorithms, PT and BDPT, with image denoising feature enabled and disabled. Then, we compared both performance and image quality results. This evaluation is described in the paper too.

Figure 1: Lift: the graphical interface

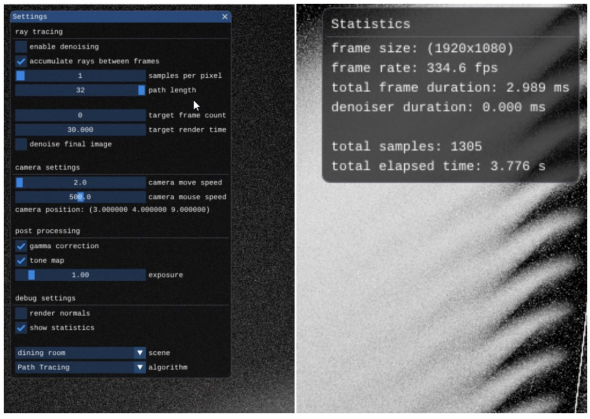

Figure 2: Tool's GUI zoom in

2 LIFT IMPLEMENTATION

2.1 Overview

The prototype was designed around a rendering architecture with progressive refinement that allows the choice of one of the two light transport techniques, the selection of the input scene and the enabling of a feature to denoise the generated image either every frame or just the final frame. Concerning the final frame of the progressive rendering, the platform's GUI allows the user to set either a target number of accumulated samples or a desired rendering's elapsed time. Statistics related with the timings, image resolution and total number of accumulated samples are provided too. Figures 1 and 2 show the tool's graphical interface with the following configuration settings: path tracer selected, denoiser disabled, 1 sample per pixel (spp), 32 bounces for the maximum path length and the rendering target defined as 30 seconds elapsed time.

The denoising step is performed by the graphics card CUDA cores exposed via the Optix framework. OptiX provides a function that takes as input noisy images generated by light transport algorithms with a small

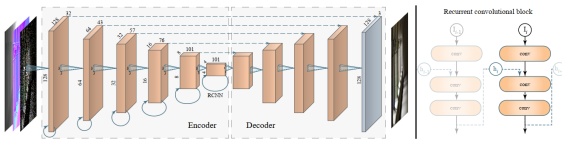

Figure 3: Architecture of the recurrent auto-encoder used in the OptiX denoiser (Chaitanya et al., 2017)

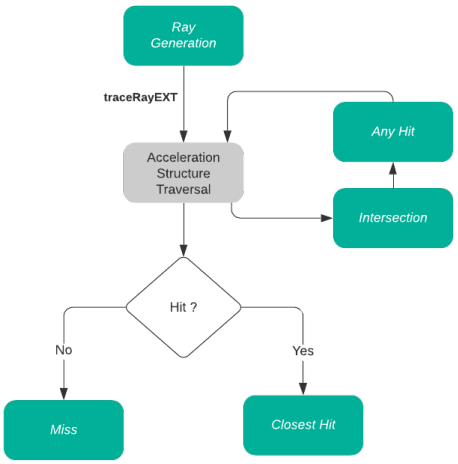

Figure 4: Vulkan's Ray Tracing Pipeline

number of samples and displays images with much higher quality, resembling images generated with many more samples.The denoiser is based on the work of (Chaitanya et al., 2017) which uses a variation of a deep convolutional network, namely the Recurrent Neural Network represented in Figure 3. The network architecture includes distinct encoder and decoder stages that operate on decreasing and increasing spatial resolutions, respectively. A recurrent block is placed at every encoding stage. Each recurrent block consists of three convolution layers with a 3 × 3-pixel spatial support. One layer processes the input features from the previous layer of the encoder. It then concatenates the results with the features from the previous hidden state, and passes it through two remaining convolution layers. The result becomes both the new hidden state and the output of the recurrent block.

2.2 Vulkan ray tracing pipeline

In order to render images by using Vulkan's Ray Tracing Pipeline, firstly it is necessary to setup up the way the Acceleration Structures (AS) are laid out in the Device Memory. Secondly, these structures should be loaded with the geometry of the scene. Finally, it would be also needed to perform their update configuration in order to support dynamic scenes which is not our case. The pipeline behaviour depends on the GLSL coding of a set of shaders: Ray Generation, Intersection, Hit, and Miss shaders. The relationship between these shaders (green blocks) is depicted in Figure 4.

Ray Generation shaders are the entry point of the ray tracing process: they are executed for each pixel on a multi-threaded 2D grid. This shader typically initializes a ray starting at the location of the camera and in a direction given by evaluating the camera lens model at the pixel location. It will then cast the rays into the scene. Other shaders below it will process further events, and return their result to the Ray Generation shader through the ray payload record. Finally, it will write the algorithm's output to memory, e.g., the screen or a texture. Two Ray Generation shaders were coded regarding both PT and BDPT algorithms. These two shaders are detailed in the next subsection.

Intersection shaders are used to implement ray-primitive intersection algorithms, other than ray-triangle intersections which have built-in support.This provides flexibility by allowing scenes with custom primitives like hair or spheres. An Intersection shader was programmed to support the classical optimized sphere-ray intersection algorithm (Akenine-Moller et al., 2018).

There are two types of Hit shaders in Vulkan: one is optional and is named Any Hit, the other is called Closest Hit. An Any Hit shader is run every time any intersection is found. The Closest Hit shader is called only for the closest intersection. A typical usage for an Any Hit shader is related with alpha testing: if the ray hits a transparent point then the intersection will be discarded and the traversal continues with that ray.

We didn't use the Any Hit shaders and our Closest Hit shaders are just responsible for creating the ray payload record which stores both the geometry and material information of the intersected primitive. This record is then sent back to the Ray Generation shader.

Miss shaders are called when a ray fails to intersect with any geometry in the scene. These shaders typically return the scene's background color or a texel from an environment map.

2.3 Ray Generation shaders

Two Ray Generation shaders were GLSL coded regarding both PT and BDPT algorithms and are described below.

2.3.1 Path Tracing

With Path Tracing the color of a pixel is computed by the averaged sum of all the paths starting from that pixel and eventually hit an emissive object. The number of paths per pixel is defined by the number of samples per pixel (spp). Progressive refinement allows the user to start rendering and to watch the image as its quality improves and stop the process once satisfied. To accomplish progressive refinement on the displayed image frame, each current pixel color is accumulated with

the contribution from previous frames and then submitted to post-processing effects, like exposure, tone mapping, and gamma correction before to be stored in a texture. The PT Ray Generation shader can be represented by the following pseudo-code:

```
void main() {
  vec3 pixel_color = vec3(0);
  for(int s = 0; s < number_of_samples; s++)
  {
    origin = pixel + random_offset();
    pixel_color += traceEyePath(origin,direction);
  }
  pixel_color /= number_of_samples;
  accumulateWithPreviousFrames(pixel_color);
  applyPostProcessing(pixel_color);
  storePixel(pixel_color);
}
```

The contribution of a path is calculated by the function ***traceEyePath***, which starts a path based on both origin and direction data inputs. Importance sampling, Next Event Estimation (NEE) and Russian Roulette techniques were deployed in its implementation. At the closest intersection, it's firstly checked if the material is emissive resulting, in that case, the termination of the path and the return of the emissive color. Otherwise, the Bidirectional Scattering Distribution Function (BSDF) of the material is importance sampled in order to calculate the direction of the path's next bounce. Then, the NEE is implemented by shooting a shadow ray from the current intersection towards an importance sampled point in a luminary or emissive object. In terms of luminaries, area lights are supported in our prototype. If there is no intersection, it means that the hit point is directly illuminated and therefore a contribution is added to that path. This contribution is weighted by the reciprocal of the probability density function (pdf) of the chosen luminary importance sampling operation. Finally, further extending of the current path is determined by the Russian Roulette Termination method. Russian roulette allows us to randomly stop computing terms in a sum as long as we reweight the terms that are not terminated. This implies the definition of a termination probability q based on the path throughput weight.Thus, the contributions of the surviving bounces will be increased by a factor $1/q$ in order to compensate the terminated bounces. The code of the function ***traceEyePath*** is listed below:

```
vec3 traceEyePath(vec3 origin, vec3 direction) {
  vec3 color = vec3(0);
  for(int bounce=0;bounce < path_length;bounce++){
    traceRay(scene, origin, direction);
    if (Hit is Emissive)
        return color * emissive;
    sampleBSDF();
    if (Ray missed) return env_color;
    //Next event estimation
    bool in_shadow = traceShadowRay();
    if (not in_shadow)
        color += material_clr / pdf;
    russian_roulette_termination();
  }
  return color;
}
```

2.3.2 *Bidirectional Path Tracing*

When compared to the Path Tracing algorithm, BDPT traces paths from the luminaries in addition to paths from the camera and connects the paths with shadow rays. The BDPT Ray Generation shader is similar to the PT one. The main difference consists of tracing, in first place, the light paths from the luminaries and performing a random walk into the scene. At each intersection, it is recorded the vertex position, the material as well as the pdf associated with the sampling of the material's BSDF function to determine the direction of the light path's next bounce. Next step consists of tracing the camera paths. And finally, at each vertex of a camera path, we connect it to all the vertices of a light path by using shadow rays: if no object is occluding the two vertices, the weight of this edge is properly computed with Multiple Importance Sampling (MIS) technique by using the balance heuristic estimator from (Veach and Guibas, 1995), and divided by the corresponding pdf. The pseudo-code for the BDPT Ray Generation shader is listed below.

```
void main() {
  vec3 pixel_color = vec3(0);
  for(int s = 0; s < number_of_samples; s++)
  {
    origin = pixel + random_offset();
    buildLightPath();
    pixel_color += traceEyePath(origin,direction);
  }
  pixel_color /= number_of_samples;
  accumulateWithPreviousFrames(pixel_color);
  applyPostProcessing(pixel_color);
  storePixel(pixel_color);
}
```

The function ***buildLightpath*** builds the light paths from the luminaries which will be used in the ***traceEyePath*** function to combine with the eye paths, as listed below:

```
vec3 traceEyePath(vec3 origin, vec3 direction) {
  vec3 color = vec3(0);
  for(int bounce=0;bounce < path_length;bounce++){
    traceRay(scene, origin, direction);
    if (Hit is Emissive)
        return color * emissive;
    sampleBSDF();
    if (Ray missed) return env_color;
```

```
    for (int l = 0; l < LIGHTPATHLENGTH; l++) {
     bool in_shadow=traceShadowRay(light_paths[l]);
     if (not in_shadow)
       color += material_clr / pdf / MIS weight;
    }
    russian_roulette_termination();
  }
  return color;
}
```

2.4 OptiX - Vulkan interoperability

The denoiser uses Cuda and rendering is done with Vulkan. The OptiX – Vulkan interoperability was built through image sharing buffers with semaphore synchronization.

The OptiX denoiser is using Cuda, so the rendered image will need to be shared between Vulkan and Cuda. The denoiser actually requires linear images, which are not available to Vulkan. So instead of directly sharing the images, we create buffers which are shared between Vulkan and Cuda and we copy the images to the buffers, converting them to linear images.

A semaphore strategy was used to tackle the problem of synchronization between denoising and rendering tasks. We use a Vulkan timeline semaphore to signal when the ray traced image is rendered and transferred to the buffer and a Cuda wait semaphore to hold the execution of the denoiser on the GPU. We add the inverse process at the end of the denoiser with a Cuda semaphore signaling the image is denoised and on Vulkan a wait semaphore to copy the buffer back to an image, tonemap it and display.

3 EVALUATION AND RESULTS

3.1 Testing Methodology

The evaluation of Lift tool was performed by using a set of scenes with different types of BSDFs in different lighting conditions in order to encompass a diversity of optical effects simulation like specular and glossy reflections/refractions, color bleeding or caustics reflections. The used scenes are illustrated in Figures 5, 6, 7 and 8 which were rendered in our prototype.

The *Teapot Cornell Box* is a simple scene with few primitives, featuring the classic Cornell Box with a teapot and showcasing different types of materials such as diffuse, glass, and colored metal.This way, optical effects like color bleeding, glossy reflection and refraction as well as caustics can be tested. The *Covered Dragon* scene includes the same classic Cornell Box too but we added a rectangular area to cover the ceiling luminary in order to make it difficult to reach. To raise the scene's geometric complexity, the known Standford Dragon, with approximately 5,500,000 triangles and

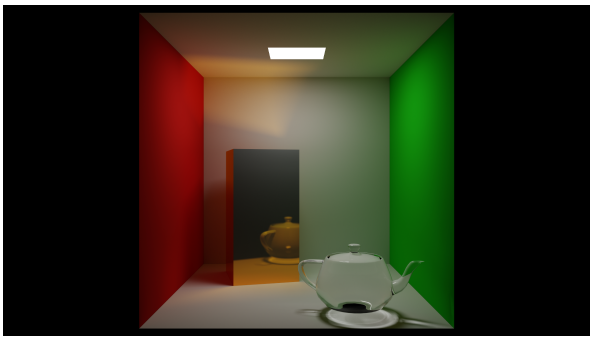

Figure 5: Teapot Cornell Box

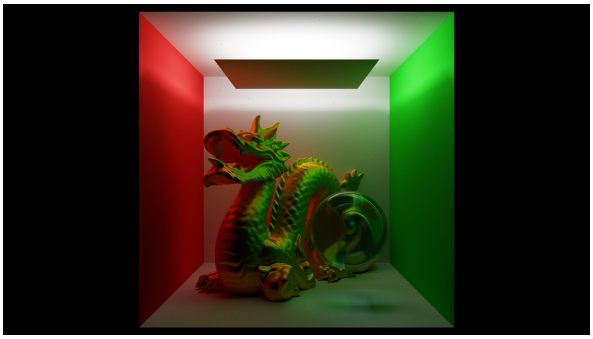

Figure 6: Covered Dragon

Figure 7: Breakfast Room (Bitterli, 2016)

Figure 8: Japanese Classroom (Bitterli, 2016)

a sphere were added. The *Breakfast Room* (Bitterli, 2016) scene features multiple different geometries and includes a lightsource placed outside the room which is illuminated through a window with blinders. Finally, we chose the *Japanese Classroom* (Bitterli, 2016), a commonly used scene to benchmark ray tracing applications and denoisers.

Both qualitative and quantitative metrics were used to benchmark the PT and BDPT algorithms. Regarding the qualitative metric to make the final image quality assessment, Structural Similarity Index Metric (SSIM) (Zhou Wang et al., 2004) is the preferred method for Monte Carlo rendered images (Whittle et al., 2017). We used a tool that computes (dis)similarity (DSSIM) between two or more PNG images using the SSIM algorithm at multiple weighed resolutions (Kornelski, 2020). The returned value is $1/SSIM - 1$, where 0 value means identical image, and > 0 (unbounded) represents the amount of difference.

The rendering experiments in both ray tracing algorithms were conducted with progressive refinement by tracing into the scene, in each frame, a single random walk per pixel (1 spp) with a maximum path length of 32 bounces. The ground truth images used in the perceptual image quality assessment (Figures 5, 6, 7 and 8) were generated with the above settings and additionally with the target rendering´s elapsed time set to 90 minutes.

The specifications of the testing computer machine were a AMD Ryzen 7 2700 8-Core CPU with a base clock frequency of 3.2GHz and 16GB of DDR4 RAM, and an NVIDIA GeForce RTX 2070 GPU with 8GB of GDDR6.

3.2 RESULTS

3.2.1 Performance

The performance of both ray tracing algorithms, PT and BDPT, was measured. The chart depicted in Figure 9 exhibits the average frames per second (fps) reached by each algorithm taking into account the time to render a frame. Consult Table 1 for more figures at different resolutions. At 1920x1080 resolution, Path tracing easily accomplished real-time rates in every benchmark scene. As far as it concerns to the BDPT, despite of its algorithmic complexity and consequently lower performance figures, it was also capable to sustain real-time framerates on the testing scenes. The worst performance scenario occurred with the *Japanese Classroom* scene where BDPT only performed at 19 fps.

The Table 2 exhibits the time spent on denoising a single frame. We verified that the duration of the denoising operation is independent of both the scene and the ray tracing algorithm being only affected by the image resolution. Mostly important, real-time framerates

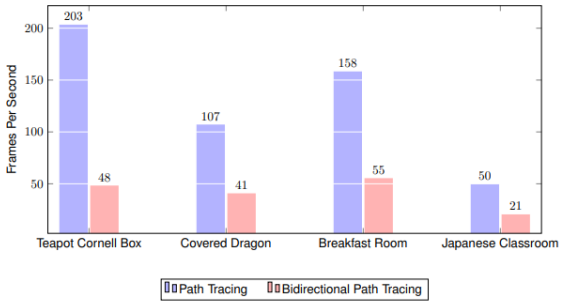

Figure 9: Average Frame Rate on 1920x1080 Resolution

	Frame Duration [ms]	
	PT	BDPT
Teapot Cornell Box 1280x720	2.3	9.6
Covered Dragon 1280x720	4.6	11.3
Breakfast Room 1280x720	2.9	8.5
Japanese Classroom 1280x720	9.3	22.2
Teapot Cornell Box 1920x1080	4.8	20.8
Covered Dragon 1920x1080	9.5	24.4
Breakfast Room 1920x1080	6.3	18.6
Japanese Classroom 1920x1080	20.2	52.4

Table 1: Frame Rendering times

were always accomplished by using the denoiser with PT algorithm and sometimes with the BDPT approach. For instance, consulting Table 1, the worst performance of PT algorithm occurred with the *Japanese Classroom* scene where the rendering of a *1920x1080* frame resolution took 20.2 milliseconds. By enabling the denoiser every frame we got 29 fps which is consistent with the duration of 14.5 milliseconds for the denoising step.

The downloaded supplemental material includes a video sequence with an animation of the movement of the camera in several scenes.

Image Resolution	1280x720	1920x1080
Denoiser Duration [ms]	7.5	14.5

Table 2: Denoiser working times

It is worth mentioning that we were able to run in Lift platform both light transport algorithms exceeding real-time rates at 1920x1080 resolution, as long as we kept a single random walk per pixel every frame. For instance, in the *Japanese Classroom* scene, by running PT algorithm with 16 spp and keeping 32 as the max path length, we achieved only 0.5 fps.

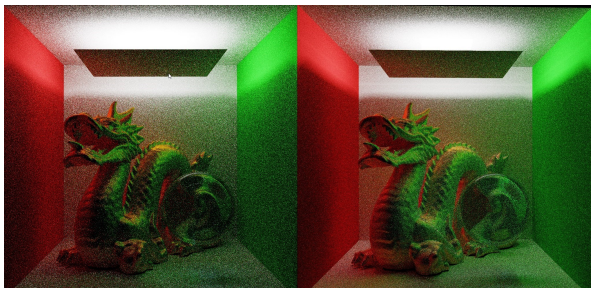

Figure 10: "Covered Dragon" image rendering after 5 seconds - Left: PT with 601 accumulated frames; Right: BDPT with 210 accumulated frames

3.2.2 *Perceptual Image Quality Assessment*

Visual inspection and convergence rates analysis were the main strategies to benchmark the perceptual quality. The progressive refinement feature allows the user to watch the image rendering as its quality improves and converges to the corresponding ground truth image. Thus, the temporal evolution of the DSSIM metric defines the convergence rate.

Firstly, we used visual inspection to compare both BDPT and PT algorithms without denoising. As expected, due to its algorithmic complexity, for the same rendering elapsed time, we got higher noisy BDPT rendered images than with PT for all scenes except for the Covered Dragon scene. As explained before, indirect illumination predominates in this scene since the luminary is covered. Thus, BDPT excels due to the combination of eye paths with light paths. Figure 10 depicts the rendered image after 5 seconds for both algorithms. We can notice that BDPT even with a lower number of accumulated samples per pixel (210 versus 601) performed better than PT algorithm.

Secondly, we analyzed how the application of the reconstruction (denoising) step affects the image quality for both light transport algorithms. This class of experiments was performed by setting the target rendering´s elapsed time to 16 milliseconds. The Figure 11 exhibits the final image frame after rendering the *Japanese Classroom* scene with PT algorithm during 16ms. This image rendering implied the accumulation of 8 frames, meaning that every pixel just received a contribution of 8 paths with a maximum of 32 bounces length each. Due to this low sample count, a significant amount of variance can be perceived in the image. Then, we repeated the experiment but now with the BDPT algorithm and the result is represented in Figure 12. It's noticeable the higher variance in the image due to the fact that, with 16ms elapsed rendering time, only 4 samples were accumulated for every pixel. Lastly, we ran again the PT algorithm but now with the denoiser set to remove the noise just in the final frame. This image frame is illustrated in Figure 13 and the very low level of variance is immediately evident when compared with the previously performed experiments. As already mentioned before, enabling the denoising step to clean the final frame implied to extend the elapsed rendering time of 14.5 milliseconds. Figure 14 reveals a side by side comparison of these images.

Figure 11: "Japanese Classroom" image rendering with PT algorithm after 16 milliseconds: 8 accumulated samples per pixel

Figure 12: "Japanese Classroom" image rendering with BDPT algorithm after 16 milliseconds: 4 accumulated samples per pixel

Figure 13: "Japanese Classroom" image rendering with PT w/Denoiser after 30.5 milliseconds - 8 accumulated samples per pixel

And finally we analyzed the convergence rates. The charts depicted in Figures 15, 16 and 17 exhibit the temporal evolution of the DSSIM metric for the *Covered Dragon*, *Breakfast Room* and *Japanese Classroom* benchmark scenes, i.e. how the rendered image quality progressively increases by accumulating samples per pixel. To conduct these experiments, we set 16384 for

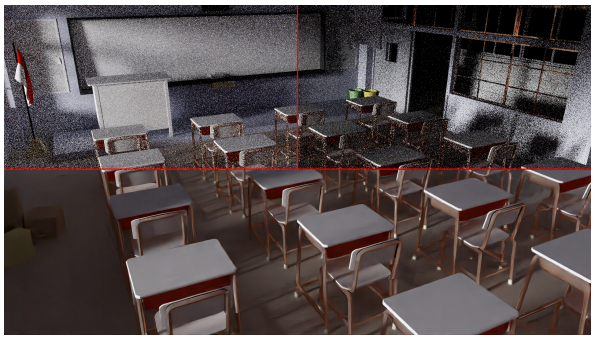

Figure 14: "Japanese Classroom" - Side by Side comparison rendered in 16 milliseconds. Path Tracing (Top-Left), Bidirectional Path Tracing (Top-Right), Path Tracing Denoised (Bottom)

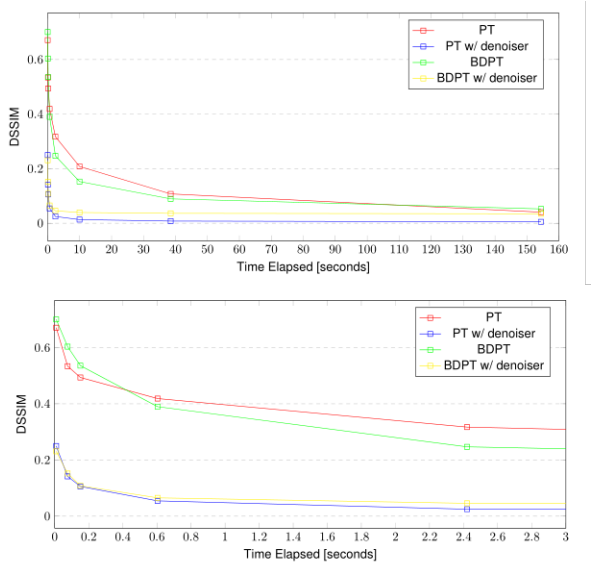

Figure 15: "Covered Dragon" - Top: DSSIM temporal evolution when accumulating 16384 frames; Bottom: Zoom in for the first 3s

the total number of accumulated frames as the rendering target.

As first observation, regardless of whether denoising operation is enabled, both light transport approaches converged to the ground truth images. This is confirmed by the decreasing trajectory of DSSIM metric over time until it gets close to 0.

A second observation, with the denoiser disabled, concerns the worst performance of BDPT algorithm compared to the unidirectional path tracer except for the Covered Dragon scene which confirmed the previous assessment done in Figure 10 with visual inspection. Looking at Figure 15 we verify that BDPT performs slightly better that PT algorithm.

As final observation, these charts clearly demonstrate that, with the denoiser enabled, the unidirectional path tracer, in all scenes, converged much faster to the corresponding reference images. Still with the denoiser on,

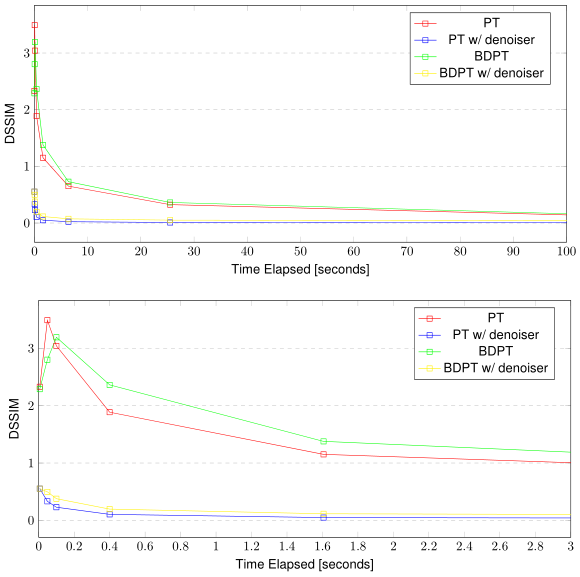

Figure 16: "Breakfast Room" - Top: DSSIM temporal evolution when accumulating 16384 frames; Bottom: Zoom in for the first 3s

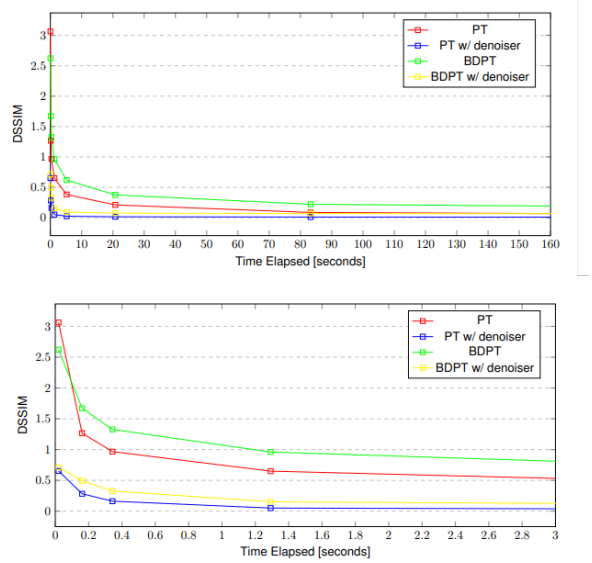

Figure 17: "Japanese Classroom" - Top: DSSIM temporal evolution when accumulating 16384 frames; Bottom: Zoom in for the first 3s

we verified that BDPT, in general, performed equally or slightly worse than PT technique.

Regarding the usage of the AI-accelerated OptiX denoiser in animation, we can notice in the video sequence, included in the supplemental material, the existence of flicker artifacts when the denoising operation occurs while the camera is moving. Several experiments lead us to conclude that this denoiser can be used for animation. However it still requires high sample counts for good results. With low sample counts, low frequency (blurry) noise can be visible in animation

frames, even if it not becomes immediately apparent in still images.

Thus, the evaluation of the two light transport algorithms in both metrics, performance and image quality, seems to indicate that, with this type of benchmark scenes, it is not worth investing in the more complex BDPT approach, because we are able to get better or similar results by using the unidirectional path tracer with denoiser.

4 TEACHING WITH LIFT

The need to develop Lift, an educational RT platform with an AI-Accelerated Denoiser, arose primarily in the context of the 3D Programming course taught at Instituto Superior Técnico (IST) and Master Theses focused on physically-based rendering. Developing a decent code base for the student projects or theses with Vulkan that could be quickly picked up was really a problem; students typically come from slightly different undergraduate studies and one could not expect the same level of competence in a programming language like C++, let alone delve into the details of the currently available ray tracing Vulkan API.

4.1 3D Programming Course

In our graduate course, 3D Programming, we supplement the theoretical lectures with 12 weekly one and half-hour laboratory sessions. Eight lab sessions are devoted to the Ray Tracing topic and include the use of Lift platform to solve the required exercises. The last four sessions regard the use of Unity3D for application development.

In the respective 8 lab classes, the students use Lift pipeline to develop two assignments which will be evaluated. In the first assignment the students should implement the Distribution ray tracer with support for glossy reflections/refractions, motion-blur and depth-of-field effects. In the second one, the students extend their ray-tracer from the previous assignment to get a Path Tracer and integrate it with the denoiser provided by Lift. Then, by using a set of benchmark scenes as well as their own scenes, the students should perform a comparison of their solutions with the built-in PT and BDPT, to assess both the performance and the perceptual image quality. In the end, the students will learn that nowadays physically-accurate images can be rendered in real-time under different lighting conditions and how well a denoiser can reconstruct images rendered with just one sample per pixel.

4.2 Master Theses

The preliminary results accomplished with Lift platform seem to favour PT with AI-accelerated Denoiser over BDPT as the best approach to obtain the highest image quality in the shortest amount of time. However, other complex ray tracing algorithms may still be able to outperform the path tracer under challenging lighting scenarios. With the Lift tool, students doing their Theses around physically-based rendering have all the programming foundations to fully answer to the question if is it worth investing on more complex algorithms, namely to perform further testing with other candidates like photon mapping-based techniques. Vertex Connection and Merging (VCM) and Unified Path Sampling (UPS) are effectively identical techniques independently developed by (Georgiev et al., 2012) and (Hachisuka et al., 2012), respectively. VCM/UPS combines bidirectional path tracing and progressive photon mapping, and is particularly advantageous for specular-diffuse paths and specular-diffuse-specular paths (i.e. caustics and specular reflections of caustics). We have currently four master students implementing over Lift platform the UPBP (unified points, beams, and paths) algorithm by (Křivánek et al., 2014) which is a generalization of VCM to volumes. It is particularly suitable for rendering volume caustics and reflections of volume caustics. Their method uses both point and beam lookups, and combines the results with Multiple Importance Sampling (MIS).

The Metropolis Light Transport-based approaches, originally proposed by (Veach and Guibas, 1997), would initially be off our radar because, in general terms, they can be seen as an extension of the bidirectional path tracer: instead of constantly creating new paths from scratch, paths are mutated into new ones. Anyway, two students are currently finalizing the evaluation of a MLT implementation over Lift and the preliminary results seem to indicate a slightly better behaviour when compared with BDPT.

5 CONCLUSIONS

In order to allow Master students, in the context of the 3D Programming course or their Theses, to address the current Real Time Ray Tracing questions formulated in section 1.2, the main objective was to build successfully an educational RT platform with an AI-Accelerated Denoiser. With Lift, we have shown to be able to accomplish interactive low-variance stochastic ray tracing. By using our prototype, the students will learn that nowadays physically-accurate images can be rendered in real-time under different lighting conditions and how well a denoiser can reconstruct images rendered with just one sample per pixel. In fact, with Lift they are able to run both light transport algorithms, PT and BDPT, in different lighting conditions exceeding real-time rates at 1920x1080 resolution, as long as we kept a single random walk per pixel every frame. We verified that the denoiser penalized the algorithms' performance by introducing a fixed reduction factor but it did not pre-

vent to achieve real-time framerates. Anyhow, this issue was largely compensated by the huge image quality improvements provided by the reconstruction filter. The OptiX denoiser revealed to own a remarkable behavior in reconstructing images rendered with just one sample per pixel. Its main drawback was the observation of flicker artifacts when the denoising operation occurs while the camera is moving. This means, that for animations, the filter is not spatio-temporal stable and our future roadmap includes its replacement by (Vicini et al., 2019) approach.

Resources: The source code repository on github is available at (Soares, 2020). The downloaded supplemental material includes a video sequence with an animation of the movement of the camera in several scenes.

6 ACKNOWLEDGEMENTS

The work reported in this article was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project UIDB/50021/2020. This work has also been kindly supported by NVIDIA.

REFERENCES

Akenine-Moller, T., Haines, E., and Hoffman, N. (2018). *Real-time rendering, Fourth edition*. AK Peters/CRC Press.

Bitterli, B. (2016). Rendering resources. https://benedikt-bitterli.me/resources/.

Burgess, J. (2020). Rtx on—the nvidia turing gpu. *IEEE Micro*, 40(2):36–44.

Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. (2017). Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):98: 1 – 12.

Daniel Koch Tobias Hector, J. B. and Werness, E. (2020). Ray tracing in vulkan. https://www.khronos.org/blog/ray-tracing-in-vulkan.

Georgiev, I., Krivánek, J., Davidovic, T., and Slusallek, P. (2012). Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, 31(6):192–1.

Hachisuka, T., Pantaleoni, J., and Jensen, H. W. (2012). A path space extension for robust light transport simulation. *ACM Trans. Graph.*, 31(6).

Kornelski (2020). Image similarity comparison simulating human perception. https://github.com/kornelski/dssim.

Křivánek, J., Georgiev, I., Hachisuka, T., Vévoda, P., Šik, M., Nowrouzezahrai, D., and Jarosz, W. (2014). Unifying points, beams, and paths in volumetric light transport simulation. *ACM Trans. Graph.*, 33(4).

Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. (2010). Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*.

Soares, G. (2020). Vulkan path tracer with optix denoiser integration. https://github.com/goncalofds/lift.

Veach, E. and Guibas, L. J. (1995). Optimally combining sampling techniques for Monte Carlo rendering. In *Proceedings SIGGRAPH'95*, pages 419–428. ACM.

Veach, E. and Guibas, L. J. (1997). Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, page 65–76, USA. ACM Press/Addison-Wesley Publishing Co.

Vicini, D., Adler, D., Novák, J., Rousselle, F., and Burley, B. (2019). Denoising deep monte carlo renderings. *Computer Graphics Forum*, 38.

Whittle, J., Jones, M., and Mantiuk, R. (2017). Analysis of reported error in monte carlo rendered images. *The Visual Computer*, 33:1–9.

Zhou Wang, Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612.