CG mandatory features reflection Group 145

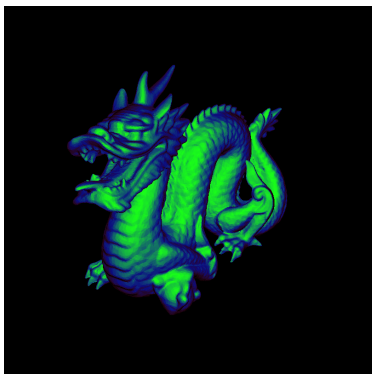
Ksawery Radziwiłowicz 6005128, Nicolae Mario-Alexandru 5988543, Shahar Katz 5958318

January 2025

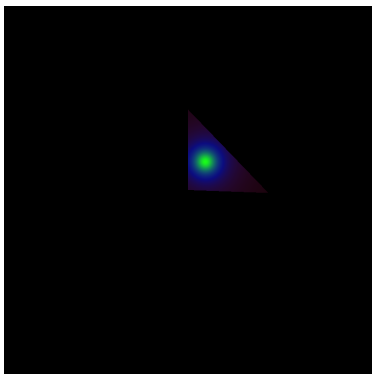
1 Shading models

1.1 Description

This feature required me to implement a custom shading model based on a linear gradient of colours. To sample a colour from a gradient, I sort the gradient's components by their t-value and find the first position where the component's t is bigger than the t I want to sample. Then I take that component and the one before and linearly interpolate between them (if there is no component with t smaller or bigger than the value sampled, I simply clamp the result to the first or last component respectively). To compute the colour on a surface based on hit information, I evaluate the diffuse Phong model, but replace K_d with the gradient sampled at point $\cos(\theta)$ where θ is the angle between the surface normal and the light ray. For the model comparison, I calculate the difference d between the colour vectors returned by Phong and Blinn-Phong models and the final colour is the result of sampling the gradient at $||d||$ if the length of the Phong vector is longer and $-||d||$ otherwise.



(a) Stanford dragon shaded with the linear gradient [(0, RED), (0.5, BLUE), (1, GREEN)]



(b) A triangle with a light source right next to it shaded with the same gradient

1.2 Reflection - linear gradient comparison

For the linear gradient comparison, the minimal number of colours I propose is 3 - 2 constrasting colours at -1 and 1 (e.g. red and green) and a dark colour at 0 (e.g. dark gray). While it is possible to see the differences without the colour in the middle, making this colour gray (instead of yellow interpolated from red and green) makes them more visible.

(a) Utah teapot shaded with linear comparison with 2 components

(b) Same picture with gray colour added at t=0 in the gradient

1.3 Location

This feature is implemented in the file `src/shading.cpp` in functions:

- `LinearGradient::sample`
- `computeLinearGradientModel`
- `computeLinearGradientModelComparison`

2 Recursive ray reflections

2.1 Description

The `generateReflectionRay` function returns a new ray that is reflected upon the normal of the mesh at the intersection point with an offset to not have a self-intersection. `renderRaySpecularComponent` function is recursively called by and calls `renderRay` each time with a higher *rayDepth* (till it reaches the base case of 6) to add the color of the recursively reflected rays.

Debug

The debug draws the ray being shot as blue, the first reflection as red, and every next reflection more and more light blue from red. Each reflection point also has a green normal ray.

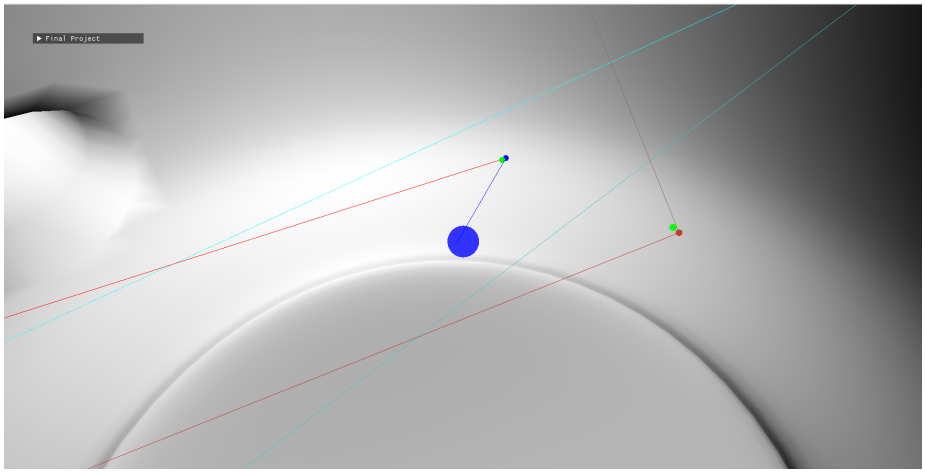


Figure 3: Image visualizing the above described debug.

2.2 Rendered image

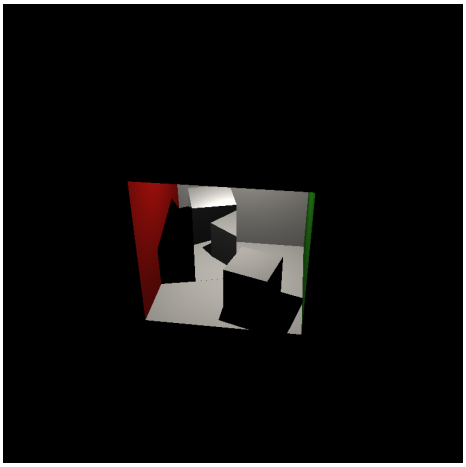


Figure 4: Reflection on Cornell Box.

Figure 5: Reflection on Teapot.

2.3 Location

This feature is implemented in the file `src/recursive.cpp` in functions:

- `generateReflectionRay`
- `renderRaySpecularComponent`

3 Recursive ray transparency

3.1 Description

The `generatePassthroughRay function` returns a new ray that is the same direction as the original and starts at the intersection point with an offset so as not to have a self-intersection. `renderRayTransparentComponent` function is recursively called by and calls `renderRay` each time with a higher *rayDepth* (till it reaches the base case of 6) to multiply the hit color light by *transparency* and the light past hit by (1-*transparency*), added to do an alpha blend.

Debug

Intersected rays are drawn as the *hitColor* of the object they hit, with a sphere of the same *hitColor* at the intersection point. The first $0.05f$ of the normal ray is colored as the *kd* value and the next $0.05f$ of the normal show gray-scale *transparency* values (the whiter it is, the larger *transparency* is).

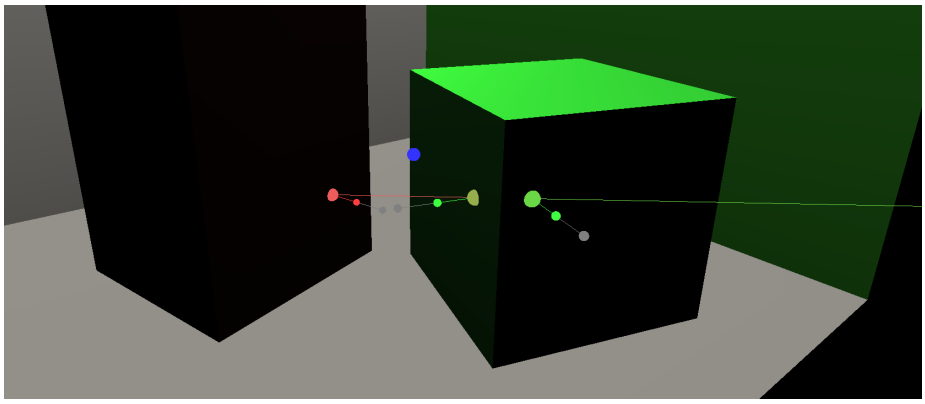


Figure 6: Image visualizing the above described debug.

3.2 Rendered image

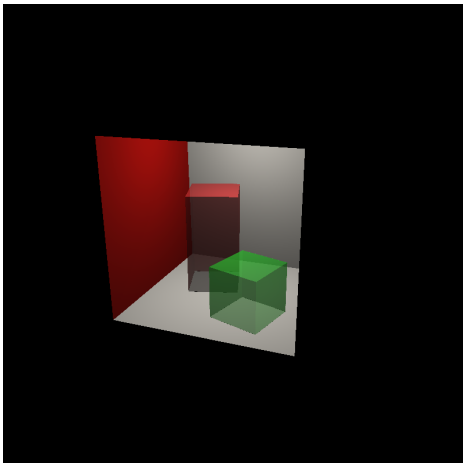


Figure 7: Transparency on Cornell Box.

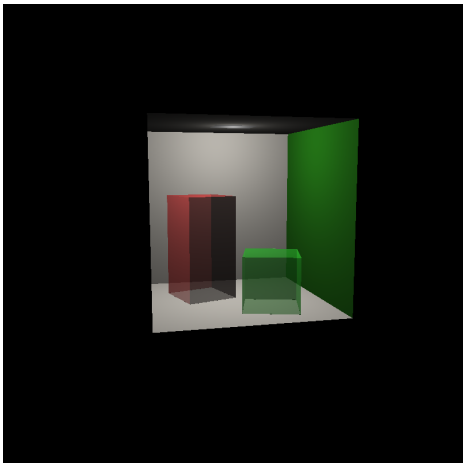


Figure 8: Transparency on Cornell Box.

3.3 Location

This feature is implemented in the file `src/recursive.cpp` in functions:

- `generatePassthroughRay`
- `renderRayTransparentComponent`

4 BVH Traversal

4.1 Description

The function `intersect` first checks for intersections in all spheres. It then uses a stack to traverse the *BVH*. It either adds the children of a node if it is not a leaf and it's *AABB* is intersected, or checks through intersections of the primitives if it is a leaf. Updating *hitInfo*'s material, barycentric coordinates, texture and normal if intersected.

Debug

The debug draws the *AABB*'s of the nodes that are being checked. The deeper the nodes the more green and less red the draw becomes. All triangle points that the method checked for intersections have blue spheres on them. The final *AABB* with the intersected triangle, and the final interested triangle is colored in white. The final intersected point is also blue.

4.2 Rendered image

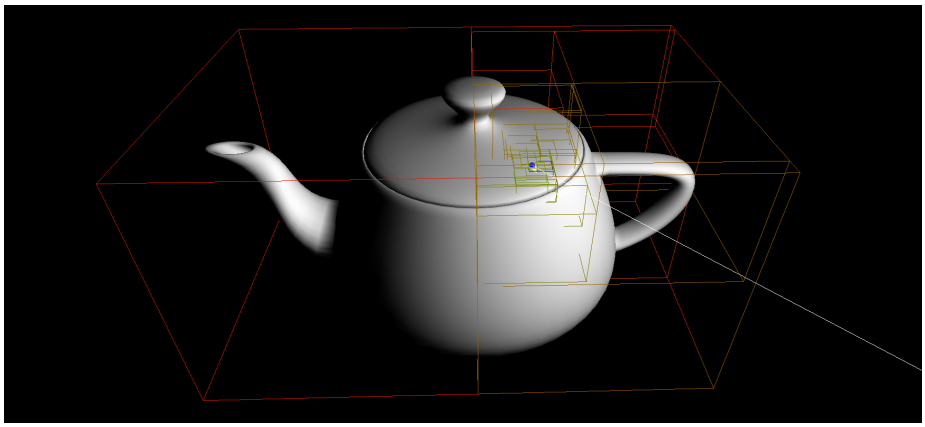


Figure 9: Working BVH on Teapot.

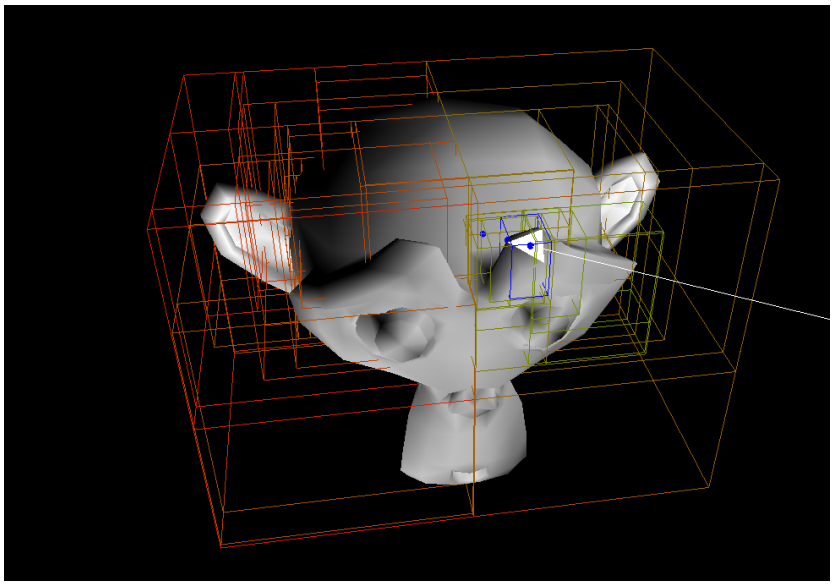


Figure 10: Working BVH on Monkey.

4.3 Location

This feature is implemented in the file `src/bvh.cpp` in functions:

- `intersect`

5 Interpolation

5.1 Description

The function `computeBarycentricCoord`, for all combinations of two vertices of the triangle, calculates the area with the point p and divides by the total area to get the alpha, beta and lambda of barycentric coordinates. The functions `interpolateNormal` and `interpolateTexCoord` take the normal and texture coordinates, respectively, and multiplies by the alpha, beta, and lambda of barycentric coordinates.

5.2 Rendered image

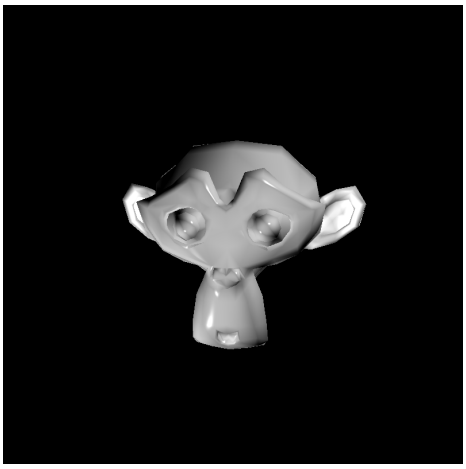


Figure 11: Working interpolated normals on monkey.

Figure 12: Working interpolated normals on teapot.

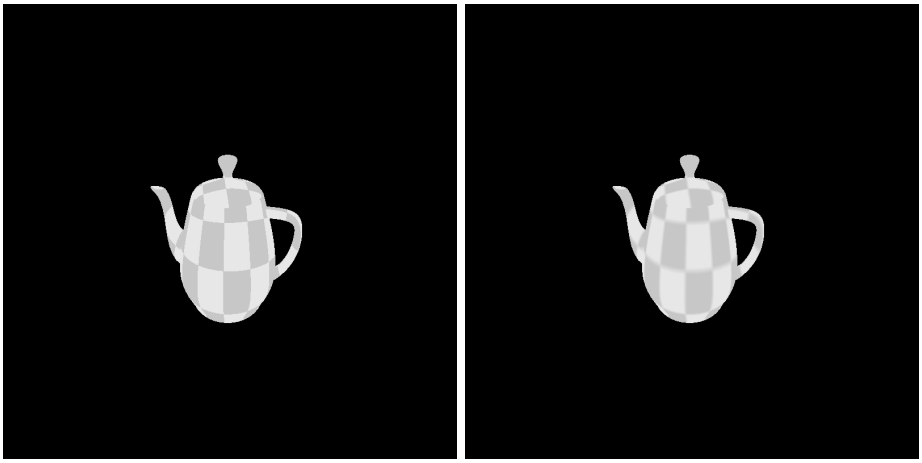
5.3 Location

This feature is implemented in the file `src/interpolate.cpp` in functions:

- `computeBarycentricCoord`
- `interpolateNormal`
- `interpolateTexCoord`

6 Texture mapping

This feature required me to implement texture mapping. This was used to enhance the visual detail of rendered images. The functionality utilizes texture coordinate data to map these textures onto 3D surfaces. I developed 2 methods: sampleTextureNearest and sampleTextureBilinear. The first one converts the texture's coordinates to a linear index and returns the pixel from that index, which shows a sharp texture appearance. The other one uses bilinear interpolation between the four nearest texels. This results in a smoother and much better texture appearance.

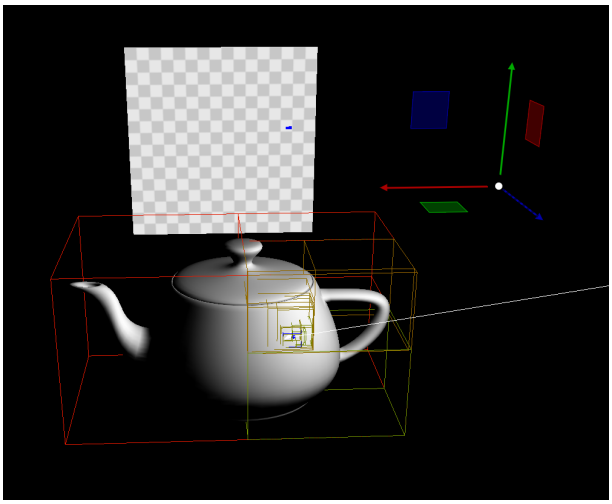


The first picture is shown after selecting BVH, Textures and pressing Render to file. The second picture is shown after selecting BVH, Textures, Texture Filtering and pressing Render to file. This feature is implemented in the file `src/texture.cpp` in methods:

- `sampleTextureNearest`
- `sampleTextureBilinear`

`Visual Debug of Texture Mapping` : I dynamically displayed a 2D representation of the texture in the 3D scene, also showing the interaction of rays with this texture, using a blue dot.

The debugging view is generated using OpenGL to render the texture and highlight the intersection point.



In the application, if you check the boxes for BVH, Texturing, Enable Debug Draw and press r on the teapot, the result from above will appear, with the texture in the background and the point where the rays touches the texture.

This feature is implemented in the file `src/shading.cpp` in the methods:

- `displayTextureMapping`
- `sampleMaterialKd`

I used sampleMaterialKd to be able to use the RenderState variable and in the other method, I am calculating the pixel data and showing the texture using OpenGL.

7 Lights and shadows

7.1 Description

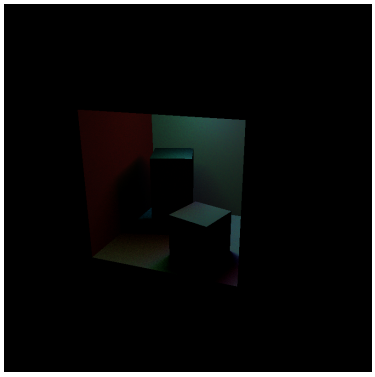
This feature focuses on implementing 3 kinds of light sources and shadows cast by opaque and transparent objects.

The first part was sampling position and colours from segment and parallelogram lights. For the former, simple linear interpolation can be used. For the latter, I bilinearly interpolate the colours, while for the position I add a fraction of `edge01` and `edge02` to `v0` based on the values of the argument `sample`.

For `visibilityOfLightSampleBinary` I calculate the coordinates of the intersection point between camera ray and the scene and between a ray from light in the direction of the previous intersection point with the scene. If both rays intersect the scene at the same point and the same side of the given face (that can be checked by comparing the signs of the cosine between camera/light ray and surface normal; they're the same iff both rays hit the same side of the surface), I return true and otherwise false. For the version of this function

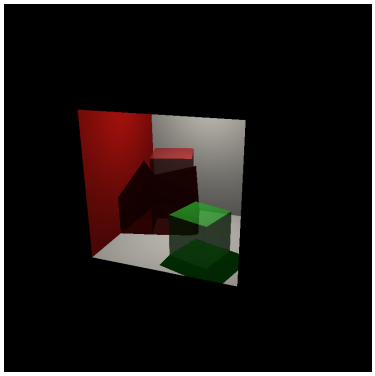
with transparency, I recursively cast rays from encountered (semi-)transparent surfaces with their colours changed according to the provided alpha blending formula. It's important to notice that the recursive rays must be offset by a small distance (10^{-6} in my case) along their direction to avoid the recursive call finding intersection in the same place. Both of those functions also include a visual debug functionality that draws the rays from the light source towards the intersection point and sets their colour to the colour of the light along the given segment if they reach their destination and black otherwise.

Finally, the `computeContributionPointLight` function uses `visibilityOfLightSample` to calculate light's colour and passes it (along some other arguments, like light ray and camera ray) to 'computeShading'. The other `computeContribution***Light` functions sample their respective light types, invoke `computeContributionPointLight` for each of the samples and average out the results.

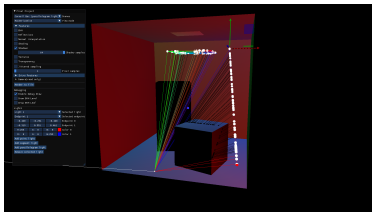


(a) The provided scene with parallelogram light

(b) Same scene but with segment light going from front right to rear left corner



(c) The provided scene with transparency



(d) Visual debug - visibility rays and samples

7.2 Reflection - sampling

If I replace `state.sampler.next_2d()` with `glm::vec2(state.sampler.next_1d(), state.sampler.next_1d())`, I get different samples. Looking at a square parallelogram light it's easy to notice that the x and y components of each sample get flipped.

7.3 Reflection - alpha blending

To illustrate why $lightColor \cdot K_d \cdot (1-\alpha)$ is incorrect, consider a fully transparent ($\alpha = 0$) surface and a light shining on it. Since the surface is fully transparent, light colour should remain unchanged; however, we clearly see that the actual light colour after passing this surface will be $lightColor \cdot K_d$ (different than light colour unless K_d is white). An improvement would be to replace the K_d term in this equation by $(1-(1-K_{dr})\alpha, 1-(1-K_{dg})\alpha, 1-(1-K_{db})\alpha)$ which will make the equation evaluate to $lightColor$ if $\alpha = 0$ and make the surface contribute more and more colour as α increases.

7.4 Location

This feature is implemented in the file `src/light.cpp` in functions:

- `sampleSegmentLight`
- `sampleParallelogramLight`
- `visibilityOfLightSampleBinary`
- `visibilityOfLightSampleTransparency`
- `computeContributionPointLight`
- `computeContributionSegmentLight`
- `computeContributionParallelogramLight`

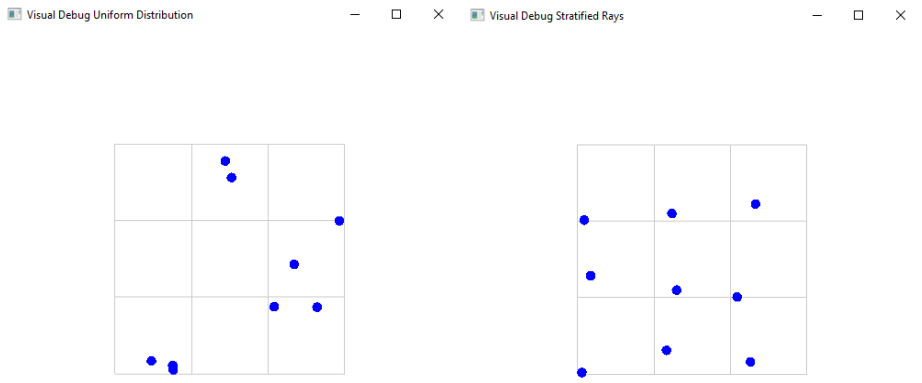
8 Multisampling

This feature required me to implement Multisampling. This involved the generation of multiple rays per pixel. I developed 2 methods: generatePixelRaysUniform and generatePixelRaysStratified. The first one makes rays uniformly within each pixel. The second one introduces jittered sampling. Here, the rays are placed at random positions within subdivided areas of each pixel.

The first picture is shown after pressing the key 'r'. The second picture is shown after selecting Jittered Sampling and pressing r. This feature is implemented in the file `src/render.cpp` in methods:

- `generatePixelRaysUniform`
- `generatePixelRaysStratified`

`Visual Debug of Multisampling` : I showed the distinct sample distributions within a pixel, showing both the uniform and stratified methods. In uniform sampling, samples are spread with gaps, whereas in stratified sampling, they are positioned in the subdivided pixel regions, promoting randomness and reducing aliasing artifacts. I made an OpenGL grid, where sample points are marked with blue. The pictures highlight the effectiveness of jittered sampling in creating a set of points that can capture better detail and reduce noise in images.



In the application, if you check or not the Jittered sampling, press r on the teapot to create the rays, check Enable Debug Draw and press r again, the result from above will appear, the pop-up window with the grid and the rays displayed on it. This feature is implemented in the file `src/render.cpp` in the methods:

- `generatePixelRaysUniform`

- `generatePixelRaysStratified`

Visual Benefits: These visualizations depict the difference in sample distribution. On one hand, uniform sampling provides a stable, but less accurate capture of fine details, making it less ideal for high quality renders, but faster for scenarios where speed matters. On the other hand, stratified sampling can lead to more accurate representation of scene details at the cost of noise and time.