Standard Features Report

Group 153
Radu Zidaru, Bo Kraijnbrink, Vasil Georgiev

Contents

1 Shading Models

1.1 Description

In this feature we implement a custom non-photorealistic shading model that is based on a gradient function rather than the material of the object.

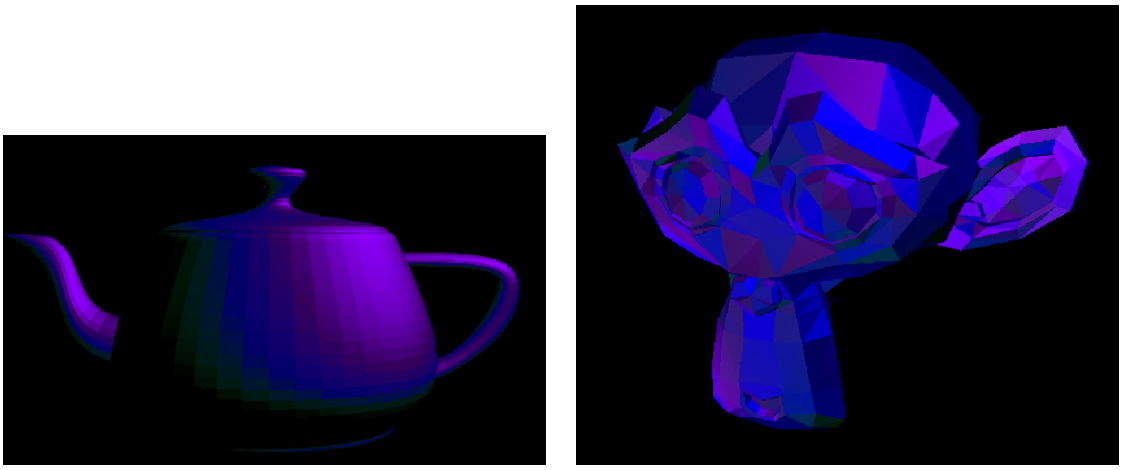
The method **LinearGradient.sample** takes a value between -1 and 1, and linearly interpolates the 2 closest components of the gradient, that is chosen.

The **computeLinearGradientModel** method calls the sampling function using the cosine value of the angle between the light direction and the normal of the hit point.

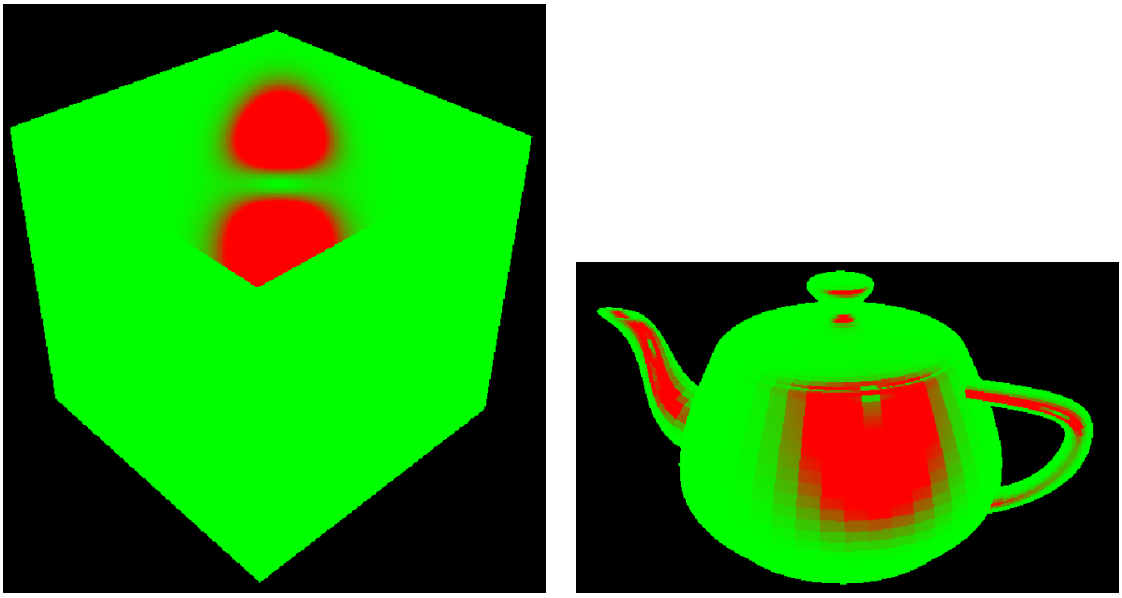
1.2 Visual Debug

The **computeLinearGradientModelComparison** method compares the Phong and Blinnphong shading models and outputs a different color, based on the difference. I would recommend using a minimum of 3 gradient components to cover the 3 distinct cases: either Phong or Blinnphong contributes more, or they contribute equally. In terms of colors, I recommend just picking 3 colors with high contrast. In this case I have used red, blue, and green.

1.3 Rendered Images



Linear Gradient



Gradient Comparison

1.4 Location

File: *shading.cpp*
Methods: **LinearGradient.sample**, **computeLinearGradientModel**, **computeLinearGradientModelComparison**

2 Recursive Ray Reflections

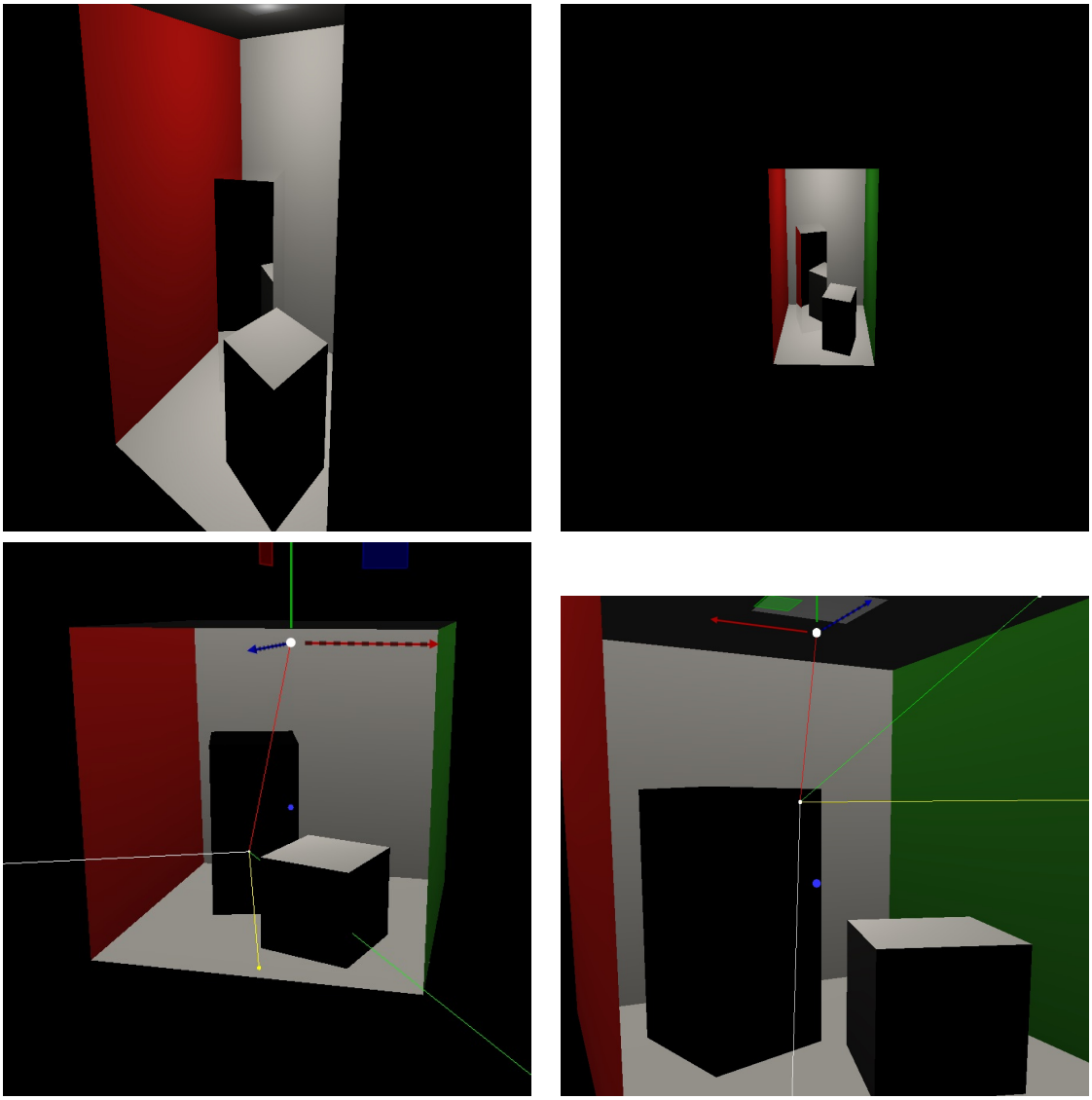
2.1 Description

In the **generateReflectionRay** method, we used the reflected ray formula from the lectures and returned the result as a Ray object, setting the right origin and direction.

In the **renderRaySpecularComponent**, we called the previous method to get the reflected ray and updated the **hitColor** parameter by adding the color of the reflected ray multiplied by the Ks of the intersection point.

We also provided debug visualization of the feature, drawing the normal vector (yellow), reflected ray (green) and light direction vector(s) (red).

2.2 Rendered Images



2.3 Location

File: *recursive.cpp*
Methods: **generateReflectionRay**, **renderRaySpecularComponent**

3 Recursive Ray Transparency

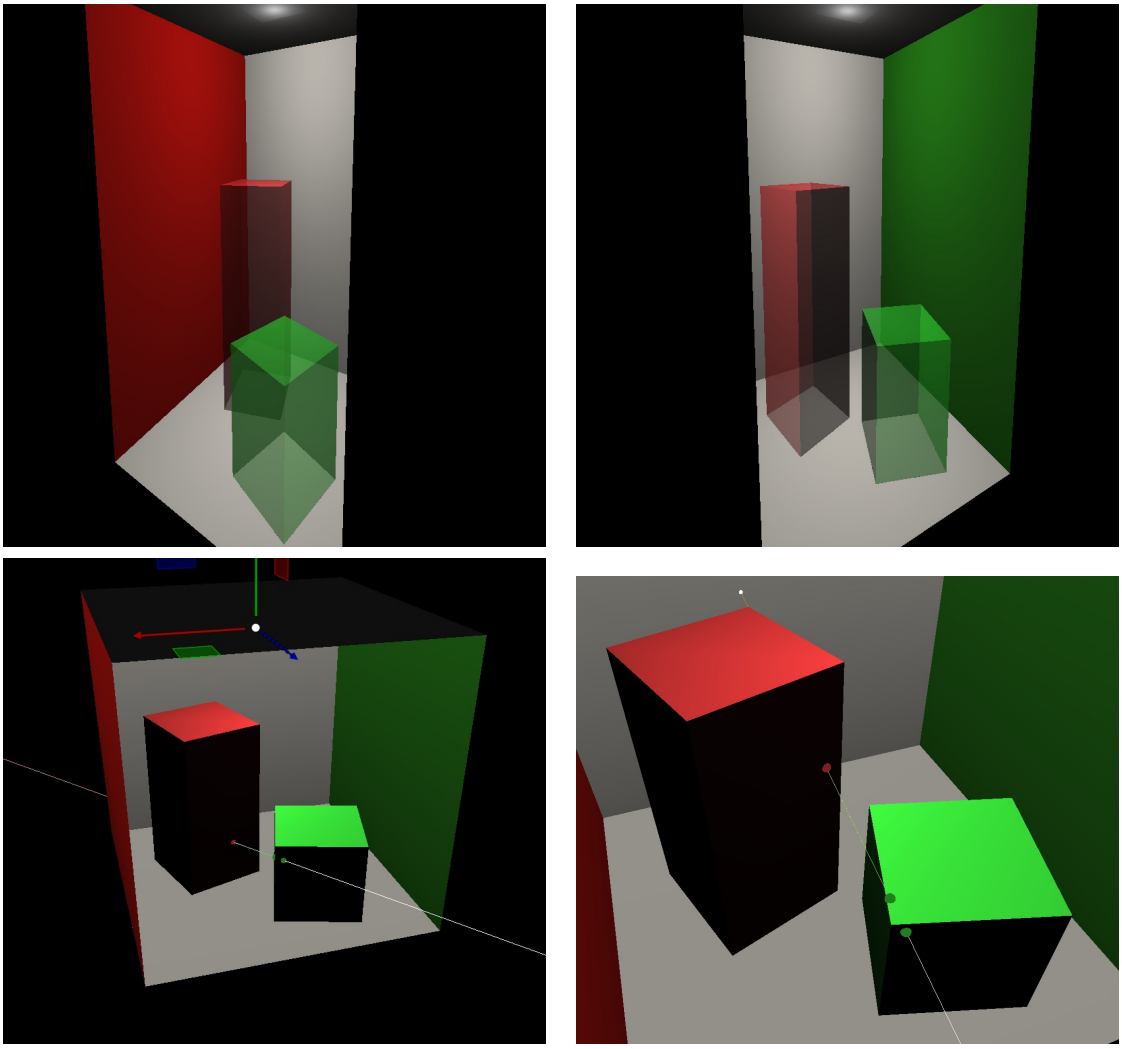
3.1 Description

In the **generatePassthroughRay** method, we simply set the origin of the resulting ray as the intersection point of the input ray with the geometry and give it the same direction.

In the **renderRayTransparentComponent** method, we use the previous method to get the pass-through ray and update the **hitColor** parameter with the alpha-blending method, using the **hitInfo.material.transparency** parameter.

Furthermore, we draw this ray in the resulting color and we also represent the Kd and transparency value visually by drawing a sphere at the intersection point with geometry, in the color of the Kd parameter, multiplied by the transparency value of the material, essentially giving us an alpha blending between the Kd color and the color black. So, the more transparent the material is, the blacker the sphere gets and the less transparent it is, the more is the color closer to the Kd value.

3.2 Rendered Images



3.3 Location

File: *recursive.cpp*
Methods: **generatePassthroughRay**, **renderRayTransparentComponent**

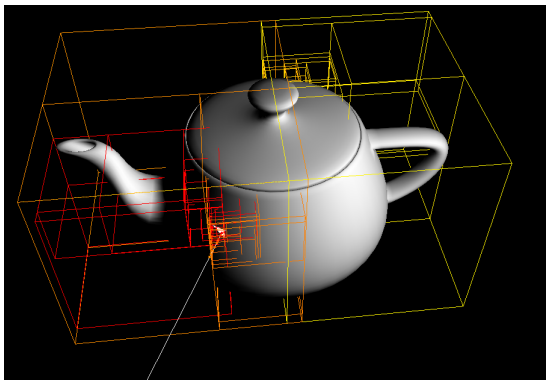
4 BVH Traversal

4.1 Description

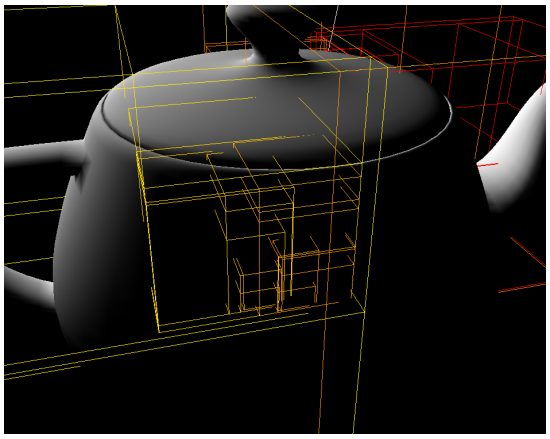
In the intersect method, we check if the BVH flag is enabled. If it is not, we use the library BVH traversal. If it is, we use our method, **intersectRayWithBVHImplemented**. This method uses a stack-based traversal to iterate over the acceleration data structure. For each node at the top of the stack, we check if our ray intersects the respective AABB using the **intersectRayWithAABB** method. If the node is a leaf, then we check if the ray intersects any geometry within that node and if it does, we check if the geometry that was hit is closer to the ray than previous hit geometry. After the stack is empty, we update the **hitInfo** object accordingly using the **updateHitInfo** method, giving this function the closest intersected geometry to the ray origin as parameter. If the node is not a leaf, then we just push its children to the stack. When we are visiting a node, we pop it from the stack.

Within the same traversal, if the visual debug flag is enabled, we draw each intersection point of the ray with the nodes and the corresponding AABBs, using a gradient ranging from yellow (visited the earliest) to red (visited the latest). The triangles that are intersected by the ray are also drawn.

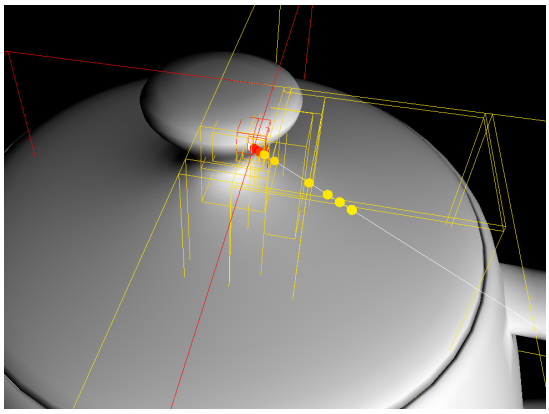
4.2 Rendered Images

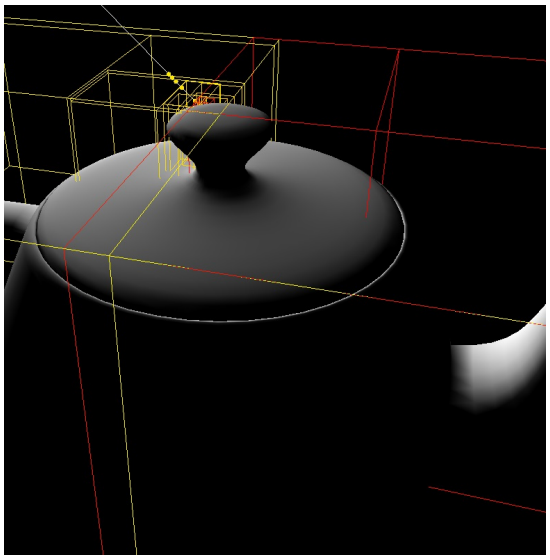


First Intersected Triangle Drawn



Second Intersected Triangle Not Drawn





4.3 Location

File: *bvh.cpp*
Methods: **BVH::intersect**, **intersectRayWithBVHImplemented**, **intersectRayWithAABB**, **intersectionPointRayWithAABB**, **updateHitInfo countIntersectedAABBs**

5 Interpolation

5.1 Description

In the **computeBarycentricCoord** method, we compute the barycentric coordinates of the point p in relation to the triangle formed by v0, v1, v2. If they show that the point is indeed inside of the triangle, we return them. If the point is not inside the triangle, we find the closest point to p that is inside the triangle and return its barycentric coordinates instead.

In the **interpolateNormal** and **interpolateTexCoord** methods, we simply use the

input barycentric coordinates as weights for each one of the normals and tex coordinates, respectively.

Furthermore, we call all the 3 methods inside of the method **updateHitInfo** in *bvh.cpp* in order to update the **hitInfo** object, but only if the corresponding normal interpolation flag is enabled. Otherwise, we stick to the regular normal of the triangle.

5.2 Rendered Images

Before Interpolation

After Interpolation

5.3 Location

Files: *bvh.cpp*, *interpolate.cpp*
Methods: **computeBarycentricCoord**, **interpolateNormal**, **interpolateTexCoord**, **updateHitInfo**

6 Texture Mapping

6.1 Description

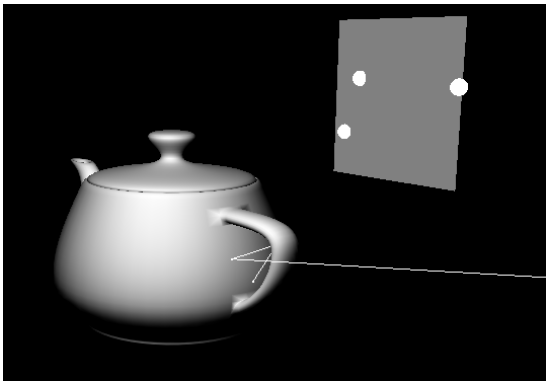
In the **sampleTexureNearest** method, we calculate the image coordinates to the texCoord, than we round that result and clamp it so it doesn't fall outside of the bounds we calculate the index of the image and return that color

In the **sampleTextureBilinear** method, we calculate the same coordinates but we floor the x and y instead of rounding and making a second x and y texel index that is plus 1 of the previous. we calculate the weight to each texel. Than we get the 4 image indexes and apply bilinear interpolation.

6.2 Rendered Images

Nearest texels texure

Interpolated texels texture



Ray intersection to texture space

6.3 Location

Files: *texure.cpp*
Methods: **sampleTexureNearest**, **sampleTextureBilinear**

7 Lights and Shadows

7.1 Description

In this feature we implement the shadow aspect of the ray tracer.
In the **computeContribution** methods we calculate the lighting contribution of point, segment and parallelogram light by sampling, checking visibility, and applying shading. They use the **sampleLight** methods to convert the samples into the intended light source.

The **visibilityOfLightSampleBinary** method checks if a light sample is visible from an intersection point, considering shadows and occlusions.

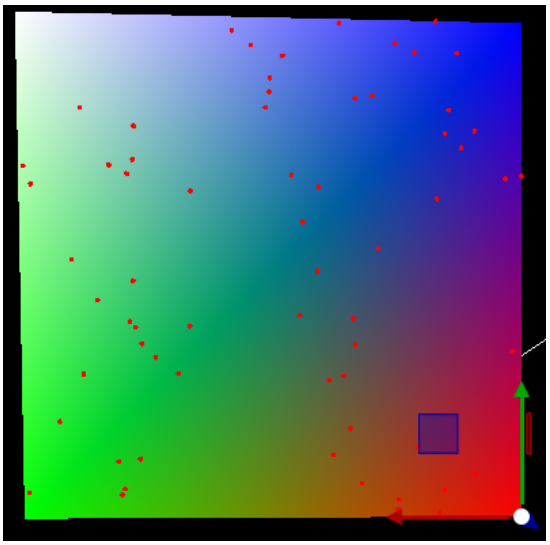
The **visibilityOfLightSampleTransparency** method calculates light visibility through

transparent objects, attenuating light color based on transparency and material properties.

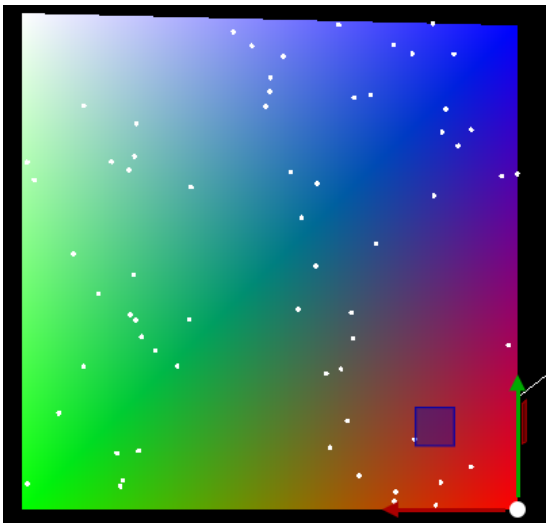
7.2 Visual Debug

There is no difference in using the **2D sample**, and using **1D sample** for both coordinates since the actual implementation of the 2D sample method uses 2 1D samplers.

The formula used in the **visibilityOfLightSampleTransparency** method is incorrect because it doesn't take into account the light that is reflected from the transparent object. To take that into account I propose the formula lightColor = lightColor * kd * (1 - transparency) + lightColor * transparency. On the first two pictures in **Renderd Images** below you can see the difference.

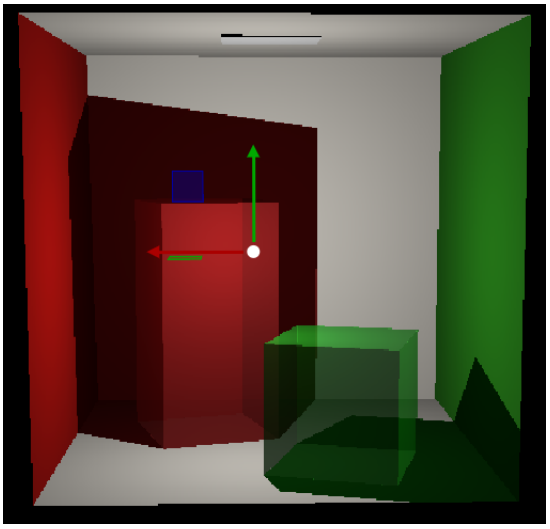


1D sample

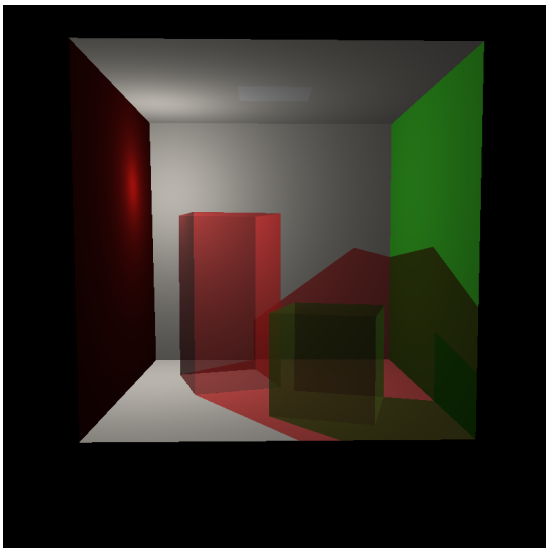


2D sample

7.3 Rendered Images

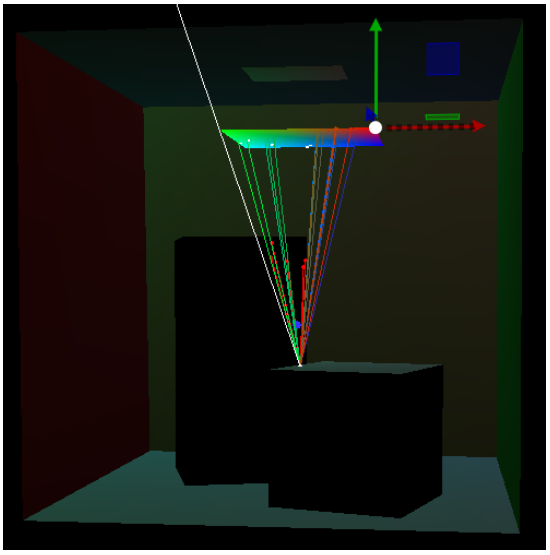


Transparency old formula

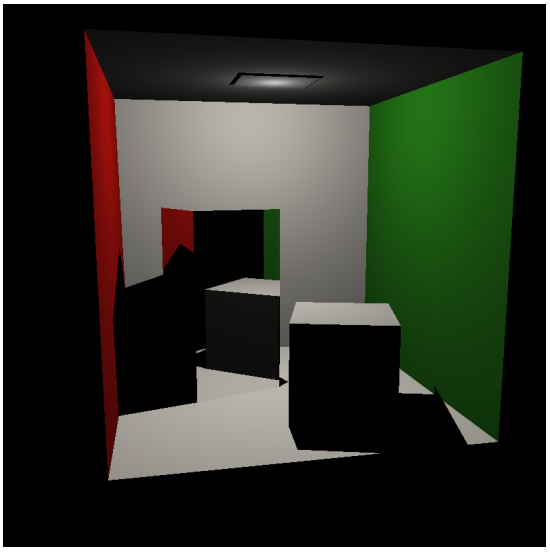


Transparency new formula

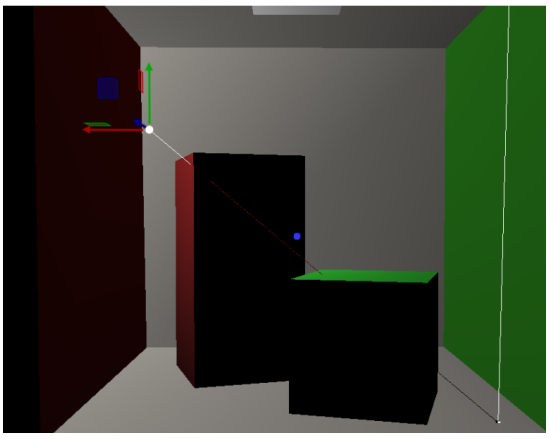
Parallelogram rendered image



Parallelogram debug rays



Mirror rendered image



Transparency debug rays

7.4 Location

Files: *light.cpp*
Methods: **sampleSegmentLight**, **sampleParallelogramLight**, **visibilityOfLightSampleBinary**, **visibilityOfLightSampleTransparency**, **computeContributionPointLight**, **computeContributionSegmentLight**, **computeContributionParallelogramLight**

8 Multisampling

8.1 Description

In the **generatePixelRaysUniform** method, it calculates the pixel size and the starting position of the pixel. Each iteration of the for loop adds a ray from the camera to the pixel staring location with a random offset times the pixel size. It does that numSamples times

In the **generatePixelRaysStratified** method, it calculates the pixel size and the starting position of the pixel. The nested for loop makes it so that we can make a grid within each grid cell we calculate the position of the ray with a random offset, only this time we divide it by the numSamples such that it stays within bounds. So we are left with numSamples * numSamples rays

8.2 Rendered Images

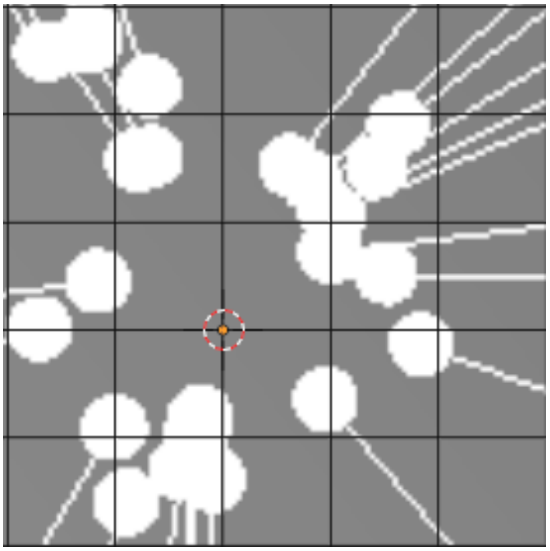
No Multi-sampling

16x Uniform Multi-sampling

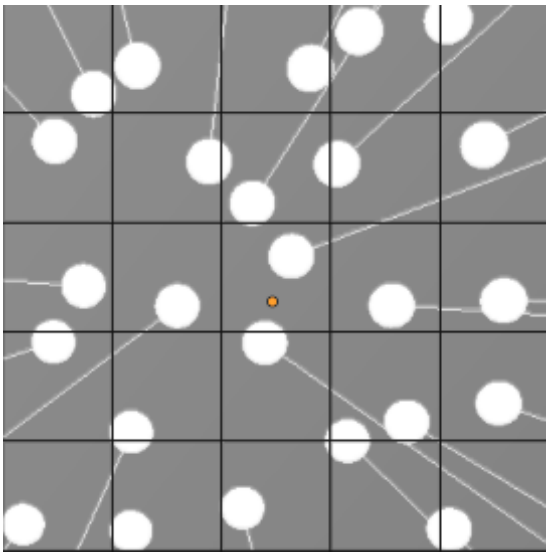
16x Stratified Multi-sampling

8.3 Reflection Assignment

8.3.1 grid visualization



Uniform Multi-sampling

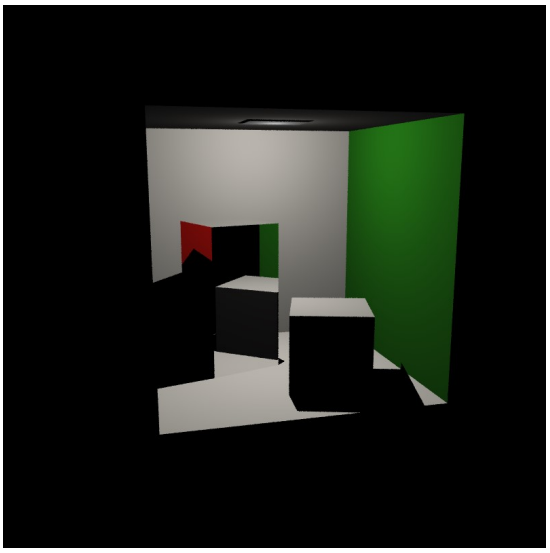


Stratified Multi-sampling

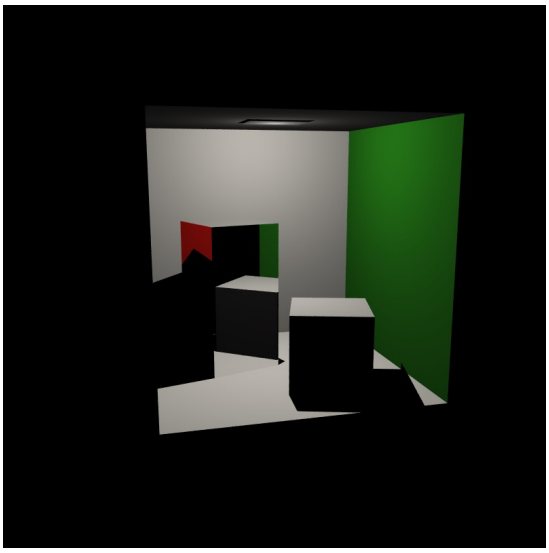
Both of the pictures shown above have a sample size of 25 rays. The difference between the two grids shown is very clear. Uniform Multi-sampling is putting all the rays in random locations within the size of the pixel. In the picture you can clearly see that there is no pattern in the Multi-sampling.

On the contrary you have Stratified Multi-sampling where with 25 rays the pixel is made up of a grid of 5 * 5 where each cell only holds 1 ray. That ray is placed within a random location in that cell. As you can see in the picture, each cell only contains one ray, but each ray is in a different location within its own cell.

8.3.2 difference visualization



Uniform Multi-sampling



Stratified Multi-sampling

Both of the pictures shown above have a sample size of 8. So you can clearly see the difference between them. As you can see Uniform Multi-sampling has a lot of noise. This

is because you have a lot of clumps of rays together which results in a poor image.

Stratified Multi-sampling will have less noise at the cost of more artifacting. This is because our implementation is jittered. So it looks more natural because it is still random, but the rays are more evenly distributed. Unjittered is generally worse, because it's more prone to having aliasing artifacts, because you removed the randomness of the pattern.

8.4 Location

Files: *render.cpp*
Methods: **generatePixelRaysUniform, generatePixelRaysStratified**