EXERCISES for GPU Path Tracing

Exercises for May classes: 2nd, 6th and 9th

The CPU and GPU tasks will be evaluated in the Assignment 1: May 13rd

The main motivation for this Assignment regarding the implementation of a ray tracer in GPU is speed. So, you will develop a GLSL implementation of a Progressive Ray Tracer. Ray tracing happens to fall into a category of problems sometimes referred to as "embarrassingly parallel". Specifically, each individual pixel can be rendered separately without knowledge of other pixels being rendered alongside it. This is just the type of work a GPU is built for and will happily churn through with relative ease. A render that could take several minutes to create a passable image at low- to medium-resolution on a CPU, can produce a higher-quality result in a fraction of the time on a GPU.

A GLSL template in **Visual Studio Code** will be available and the students can use it to build their implementations.

STRONGLY ADVISED REFERENCES:

- Basic Camera, Diffuse, Emissive: https://blog.demofox.org/2020/05/25/casual-shadertoy-path-tracing-1-basic-camera-diffuse-emissive/)
- Image Improvement and Glossy Reflections: https://blog.demofox.org/2020/06/06/casual-shadertoy-path-tracing-2-image-improvement-and-glossy-reflections/
- Fresnel, Rough Refraction & Absorption, Orbit Camera: https://blog.demofox.org/2020/06/14/casual-shadertoy-path-tracing-3-fresnel-rough-refraction-absorption-orbit-camera/

GPU RAY TRACING IN A WEEK

We will start by summarizing the changes involved to move the CPU-based implementation to run entirely on the GPU. It is assumed that students have already implemented at least a Distribution Ray Tracer and have some knowledge of GLSL.

SETUP

The students can program directly on the Shadertoy (https://www.shadertoy.com/). Shadertoy is a website that lets you write small programs in a c-like language called GLSL which are ran for every pixel in an image, on your GPU (not CPU). Shadertoy uses WebGL but manages everything except the pixel shader(s) for you.

These programs have very limited input including pixel coordinate, frame number, time, and mouse position, and give only a color as output. Despite the simplicity, people have done amazing things, like making a playable level of doom 1.

The students can also use the Visual Studio Code (https://code.visualstudio.com/download) with the following Shadertoy plugin while implementing the ray tracer:

https://marketplace.visualstudio.com/items?itemName=stevensona.shader-toy

With this extension, a student can view a live WebGL preview of GLSL shaders within VSCode, similar to shadertoy.com by providing a "Show GLSL Preview" command. The ray tracer here is written in GLSL in order to take advantage of the Shader Toy plugin and get immediate feedback while working. **A GLSL code template in Visual Studio Code will be provided to the students.**

Be aware of some minor differences between programming in Shadertoy.com and in VS Code with Shadertoy plugin. For instance, in the shader entry point the first use void mainImage(out vec4 fragColor, in vec2 fragCoord) while in VS Code it's void main() and you use gl_FragCoord and gl_FragColor.

CODE CHANGES

For the CPU ray tracer, students took advantage of C++'s concepts of inheritance and polymorphism to present a clean and simple interface for hittable objects and materials to implement. This serves to simplify much of the implementation, as the implementer can then rely on the correct method being executed based on the underlying type.

Since these commodities are not available on the GPU side, we can instead employ type identifiers when it is necessary to be able to perform an action in a polymorphic way.

```
// MT_ material type
#define MT_DIFFUSE 0
#define MT_METAL 1
#define MT_DIALECTRIC 2

struct Material
{
    int type;
    vec3 albedo;
    float roughness;    // controls roughness for metals
    float refIdx;       // index of refraction for dialectric
};

bool scatter(Ray rIn, HitRecord rec, out vec3 atten, out Ray rScattered)
{
    if(rec.material.type == MT_DIFFUSE)
    {
        // ... diffuse scatter response
        return true;
    }
    if(rec.material.type == MT_METAL)
    {
        // ... metal scatter response
        return true;
    }
    if(rec.material.type == MT_DIALECTRIC)
    {
        // ... dialectric scatter response
        return true;
    }
    return false;
}
```

RANDOM NUMBERS

C's and C++'s random number utilities make random number generation CPU-side rather straightforward. C++ provides the <random> header, and the types and functions therein can be used to easily generate random numbers in a given range. For ray tracing purposes, we are specifically interested in generating numbers between 0 and 1.

There are several implementations of GPU-based pseudo-random numbers. An implementation is provided in the file common.glsl which contains adapted hash functions from Nimitz (https://www.shadertoy.com/view/Xt3cDn). The functions **random_in_unit_disk** and **random_in_unit_sphere** are also programmed in the file common.glsl.

PROGRESSIVE RAY TRACING

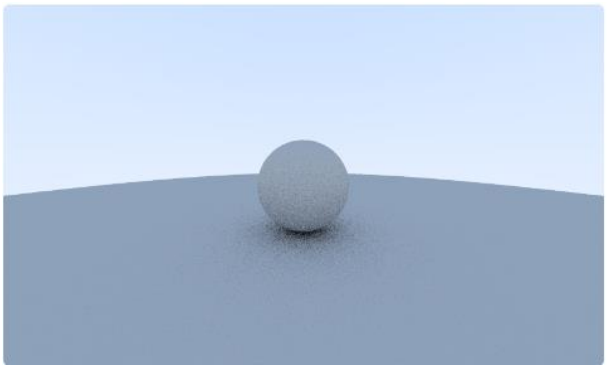
Instead of taking multiple samples per pixel in each frame, the students should create a feedback loop where all previous results are fed into the current frame. Each frame will use a random offset within the current pixel footprint by way of the noise functions mentioned earlier, and have its result appended to a running average of all previous frames. This can be done by setting in the top of the shader the following directive: #ichannel0 "self". This directive sets the previous frame's result as a texture input to the current frame. Read the Shadertoy plugin's documentation. The new shader entry point looks like the following:

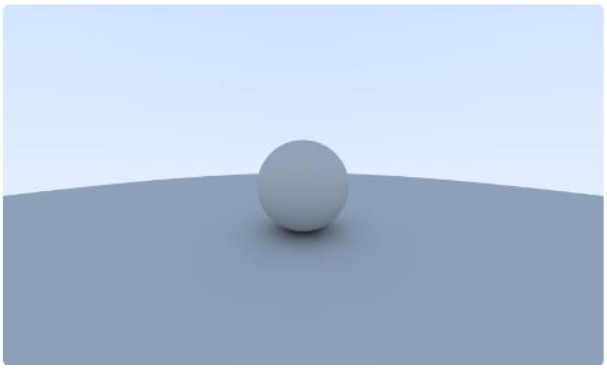
```
vec2 uv = gl_FragCoord.xy / iResolution.xy;
vec4 prev = texture(iChannel0, uv);
vec3 prevLinear = toLinear(prev.xyz);
prevLinear *= prev.w;

uv = (gl_FragCoord.xy + hash2(gSeed)) / iResolution.xy;
vec3 col = color(getRay(cam, uv));

if(iMouseButton.x != 0.0 || iMouseButton.y != 0.0)
{
    col = toGamma(col);
    gl_FragColor = vec4(col, 1.0);
    return;
}
if(prev.w > 5000.0)
{
    gl_FragColor = prev;
    return;
}

col = (col + prevLinear);
float w = prev.w + 1.0;
col /= w;
col = toGamma(col);
gl_FragColor = vec4(col, w);
```

Since the number of samples is ever-increasing, this approach has the potential to run into floating-point issues when the number of samples in the running average becomes large. If you remove the if(prev.w > 5000.0) block and allow the ray tracer to run long enough, you're likely to see little black dots show up in the image. These are caused by values becoming so high in the running average that they are no longer representable as floating-point numbers and end up as nan or inf. Capping the number of samples allows for a high-quality render and avoids these issues. The value can be adjusted up or down depending on scene and preference. There are almost certainly more robust ways to solve this issue, but in the spirit of simplicity, that will be outside the scope of this post.

Because of the substantial speedup gained by moving the work to the GPU, simple camera controls can be added to the scenes. In cases where the camera moves, the running average is reset to prevent the views smearing across each other. This is what the if(iMouseButton...) check is doing.

Below are two images of the same scene, one taken moments after the render began, the other taken after a few seconds of accumulation.





RECURSION

GLSL does not support recursive functions. This limitation is simple enough to overcome by instead using a loop with a number of steps limited by the MAX_BOUNCES.

We'll start out a pixel's color at black, a "throughput" color at white, and we'll shoot a ray out into the world, obeying the following rules:

- When a ray hits an object, (local color * throughput) is added to the pixel's color, and
- a secondary ray will be scattered which means that the initial ray will be simply overwritten with the result of the ray produced by the scattering event before the next loop iteration and the throughput of this new ray will be updated by multiplying it by the object's albedo.
- Regarding a diffuse object consider scattering a secondary ray in a random direction within a hemisphere to accomplish the color bleeding effect.
- Regarding the dieletric material, use the probabilistic maths to decide if scattering a reflected ray or a refracted ray.
- We will terminate when a ray misses all objects, or when N ray bounces have been reached.

SCENE REPRESENTATION

Instead of creating a pseudo-polymorphic hittable type as described above for materials, I've instead opted for building the scene on each invocation of a hit world function that can be implemented freshly in each shader depending on what content is desired.

Of course, there's nothing stopping you from implementing a more flexible Hittable type, similar to how materials are handled, and having a type identifier decipher which intersection function should be used. For the purposes of these exercises, the above approach works plenty.

TASKS

Download the **GPU_template.zip** which contains two GLSL files for VS Code: P3D_RT.glsl and common.glsl. Analyse and understand the code. Consult the references listed above about path tracing in Shadertoy.

1) Complete the following functions in common.glsl:
    - Ray getRay(Camera cam, vec2 pixel_sample)
    - float schlick(float cosine, float refIdx)
    - bool scatter(Ray rIn, HitRecord rec, out vec3 atten, out Ray rScattered)
    - bool hit_triangle(Triangle t, Ray r, float tmin, float tmax, out HitRecord rec)
    - bool hit_sphere(Sphere s, Ray r, float tmin, float tmax, out HitRecord rec)
    - bool hit_movingSphere(MovingSphere s, Ray r, float tmin, float tmax, out HitRecord rec)
2) In the file P3D_RT.glsl complete the functions vec3 directlighting(pointLight pl, Ray r, HitRecord rec) and vec3 rayColor(Ray r) .