

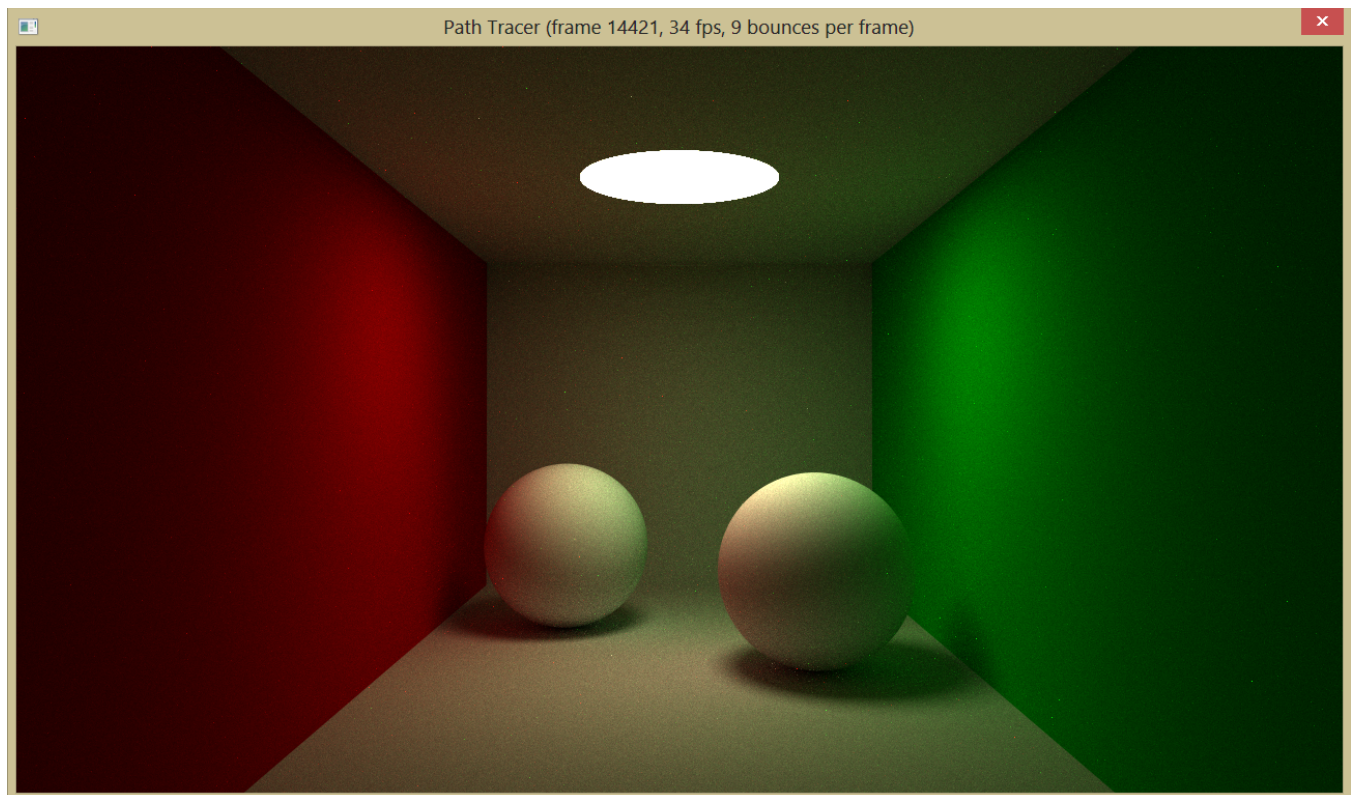
Path Tracer Practical Assignment

Advanced Graphics course

Student: Joeri van der Velden

Solid ID: 3928853

Rev. 1



Video at: <http://www.youtube.com/watch?v=IKTm2xfpZMI> (or: <http://tinyurl.com/gfx-pathtracer>)

References used for this assignment:

- Microsoft MSDN knowledge base
- http://wiki.cgsociety.org/index.php/Ray_Sphere_Intersection
- <http://stackoverflow.com/questions/6533856/ray-sphere-intersection>
- <http://sizecoding.blogspot.nl/2008/11/intersecting-ray-with-spheres-glsl.html>
- <http://mathworld.wolfram.com/SpherePointPicking.html>
- <http://gamedev.stackexchange.com/questions/32681/random-number-hlsl>

Application instructions / performance

- Use WASD to move through the scene
- Press and hold the left mouse button to look around
- Use number keys (1 to 4) to change shader mode
 - Shader mode 1 is the default scene
 - Shader mode 2 applies a mirror material to the small spheres in the room
 - Shader mode 3 is an expected value test for the pseudo-random number generator (every pixel should approach the colour 0.5, 0.5, 0.5)
 - Shader mode 4 is a simple attempt at noise removal
- Press R to reset the camera position and scene

A few notes on performance: the framerate is adaptive and targets the range 30 to 40 frames-per-second by lowering or incrementing the amount of light bounces it performs per pixel per frame, based on the current fps.

Given the 1280x720 resolution, my laptop (using its dedicated Nvidia GTX 660M graphics card) manages to hit 10 bounces per pixel per frame. My desktop (using an AMD HD7950) goes up to 60 bounces per pixel per frame.

My laptop's integrated graphics card (Intel HD Graphics 4000) didn't play very well with the adaptive framerate, it could only barely manage one bounce per pixel per frame, and even then it stayed below 30 fps. This also seemed to cause some weird rendering artefacts that I did not see with better performing graphics cards.

Additionally implemented features

Adaptive framerate

The application tries to keep scene refreshing between 30 to 40 frames per second. It does this by incrementing or decrementing the amount of light bounces the pixel shader performs when drawing a frame. More bounces means the image converges faster.

Anti-aliasing

The vertex shader passes two ray direction vectors to the pixel shader; one for the top left of the pixel, and one for the lower right. Using this information the pixel shader fires its initial ray towards a random point on the pixel's surface area. This feature is always on in the application.

Mirror material

When switching to shader mode 2 (by pressing '2'), the spheres in the scene will have a mirror-like material applied to them, by perfectly reflecting incoming rays.

Noise removal

Press 4 in the application to try out the noise removal. I was wondering if I could remove some of the pixel artefacts by applying a simple noise removal check: every pixel averages the colours of its horizontal and vertical neighbours, compares that to its own colour, and if that deviates too much it shows the neighbour average instead of its own colour. This does not change the actual paths or light gathered by that pixel, it only overrides the final colour it'll show.

In the end the difference in image quality wasn't too spectacular, and the performance suffers quite a bit. Maybe more elaborate noise removal techniques would yield better results.

Code structure description

The PathTracerWin32.cpp is the entry point of the application, at the WinMain function. Here it creates instances of the Window, Camera and Renderer classes, and then goes into the program loop until termination.

The Window instance creates the window using the Windows API and holds the window handle. This handle is passed to the Renderer instance, which initializes the Direct3D by creating the swap chain, various buffers, render target, unordered access view and viewport.

Next the Renderer initializes the pipeline by loading the compiled VertexShader and PixelShader shaders from the .cso files, while creating the layout and data buffers to pass information to the shaders.

Finally, the Renderer creates a screen-filling quad by passing two triangles to the vertex buffer.

The camera is initialized to its default position. It stores its position and rotation, and triggers scene resets if needed.

In the main program loop the current time is updated, shader data is updated and a frame is rendered. It also executes the Step() function every 1/60 of a second, to update camera movement.

Code and performance commentary

Render artefacts

When running the application for a prolonged time to let the image converge to the expected value, some pixels do not seem to converge properly. I'm not too sure what causes this. I suspect it's a limitation of the random number generator.

Scene resets

When the camera moves, resets, or the shader mode changes, every pixel needs to dump all its gathered light and start over. I do this by passing a boolean to the constant buffer which triggers a restart. It feels a bit hacky but I'm not too sure how to pass an 'event' like this to the GPU otherwise.

Ray generation

The camera passes on its inverse view projection matrix to the vertex shader through the constant buffer. In the vertex shader, two ray directions vector are generated by multiplying the inverse view projection matrix with the screenspace coordinates (which are combined with a little offset based on the pixel surface area).

In the pixel shader these two direction vectors are used to shoot a ray from the eye position towards a random point on the current pixel (which provides anti-aliasing).

Light bouncing

The maximum amount of light bounces the pixel shader performs per frame is passed by the constant buffer. Every bounce the shader first checks if the pixel is currently in a path. If not, it starts a new path from the eye position and initializes part of the ray data.

Next it loops over all the objects (spheres) in the scene and performs a ray-sphere intersection check. If the ray intersects a sphere, it calculates the point of intersection and the surface normal. Based on this a new ray is generated (using uniform sampling for diffuse materials, and reflects the ray for reflective materials) and the reflectance function combined with the weight is stored (only if the material is diffuse).

After this is done, the Russian Roulette is performed. If the path is terminated, the collected light is added to the total light and the path counter is incremented.

The current pixel colour is total light divided by the path counter.

Storing rays

The pixel shader uses a RWStructuredBuffer (made possible by creating an Unordered Access View in the Renderer) to store ray data between every bounce. To be precise it stores:

- float3 RayPos; - Current ray position
- float3 RayDir; - Current ray direction
- float3 Light; - Light collected by current path
- float3 Refl; - Combination of all the reflectance functions and weights in the path
- float3 TotalLight; - Total light collected over all paths
- float2 Seed; - Seed used for random number generation
- uint PathCounter; - Amount of paths terminated
- bool HasPath; - Whether it needs to generate a new path at the next bounce

Unbiased

Shader mode 3 demonstrates that the random number generator's expected value approximates 0.5 for all pixels. As for the rays themselves, every light bounce is weighted using the Russian Roulette termination chance to remove bias.

Pseudo random number generator in shader

The prng used by the pixel shader is probably the thing I have the least theoretical knowledge for to describe, but it seems to work. It uses a combination of CPU generated seeds (passed to the shader using the constant buffer), the previous seed, the pixel position and the time (also passed through the constant buffer), along with some internet-provided prng algorithms to generate a random number. When using the application, press 3 to see the expected value test. Every pixel should converge to the colour 0.5, 0.5, 0.5. This doesn't necessarily mean that all the values are uniformly distributed, but at least the expected value is approximated.