# rsmgpt

## Getting started

We’re going to be writing this from scratch in D3D12 to achieve two goals:

1. Learn how to build a GPU path tracer from the ground up.
2. Learn as much about D3D12 as possible in the process to assist with the day job ☺

The D3D12 samples are a pretty good starting point as they have minimal dependencies. We can use additional libraries as/when necessary.

Smallpt is also an invaluable resource as it implements path tracing in 99 lines of C++. Pretty useful to get an overall idea of how the HLSL will need to be constructed.

Let’s break down the two to see what we’ll need and how it’s going to look.

### D3D12 app structure

The sample apps extend an abstract base class called DXSample. It is structured as follows:

* Public interfaces
  + DXSample(UINT width, UINT height, std::wstring name);  
    Sets members m\_width and m\_height.   
    Parses command line args to determine whether WARP has to be enabled or not.   
    Sets m\_assetsPath to the absolute path of the exe and m\_aspectRatio to m\_width/m\_height.
  + virtual ~DXSample(); empty
  + int Run(HINSTANCE hInstance, int nCmdShow);  
    Contains the main rendering loop and the window creation code.   
    Pure virtual protected methods are invoked here.
  + void SetCustomWindowText(LPCWSTR text);  
    Appends ‘text’ to m\_title and makes the resulting string the window title.
* Protected members
  + virtual void OnInit() = 0;
    - Implementation loads all necessary resources and sets up the pipeline as necessary.
    - Create a debug layer and device.
    - Create a command queue. We’ll probably only need a compute queue to begin with but let’s see how that goes.
    - Create a swap chain and set m\_frameIndex to point to the current back buffer index.
    - Create descriptor heaps as necessary. May only need rtv and cbv/srv/uav heaps at this point.
    - Create an RTV and a command allocator for each frame.
    - Create a root signature/parameters. Will need to decide what needs to go into descriptor tables and what needs to go into root constants.
    - Compile shaders and create a pipeline state object (PSO). Probably only need one for compute for now.
    - Create a command list for execution. Note that the command list is put into recording mode as soon as it is created.
    - Create the required resources. There is a generic CreateCommittedResource method available which is used to create any type (buffer/texture(1/2/3)d) of resource along with an implicit heap to which it will be mapped. Make note of how resource barriers are used to transition resources between states when they are initialized from the CPU side.
    - Close the command list and execute it on its respective queue of the same type. The command list has to complete execution before we can proceed.
  + virtual void OnUpdate() = 0;
    - Any animation parameters and frame based updates need to be performed here.
  + virtual void OnRender() = 0;
    - Invoke the methods to populate the command lists and execute them.
    - Present the frame and wait for the command lists to finish executing.
  + virtual void OnDestroy() = 0;
    - Just wait for any pending command lists to finish executing.
  + virtual bool OnEvent(MSG msg) = 0;
    - Process any input (keyboard/mouse) events here.
  + static LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
    - Fixed implementation that can be ccp-ed from the sample code.

Should be able to base the engine implementation off of this without much of a hassle.

### Smallpt implementation

Scene description is hardcoded but we should be able to use rapidxml to implement an xml scene file parser.

Vec class which implements a vector-3 type. Should be able to use float3 and HLSL intrinsics for the dot and normalize ops. Cross product is something we may have to implement by hand.

Ray struct should be simple enough to implement as is. Need two float3’s to represent origin and direction.

Looks like an image plane only requires the width and height to be described, no need for a z position wrt the camera.

* int w=1024, h=768, samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples
* Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // cam pos, dir (z axis)
* Vec cx=Vec(w\*.5135/h) = Vec(1024\*.5135/768) = Vec(0.685) = (0.685,0,0) // cam x axis, 0.5135 is the aspect ratio.
* cy=(cx%cam.d).norm()\*.5135 = (0, 0.513, -0.022) // cam y axis
* double r1=2\*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1); r1 = 0.003, dx = -0.95
* double r2=2\*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2); r2 = 1.127, dy = 0.066
* Vec d = cx\*( ( (sx+.5 + dx)/2 + x)/w - .5) + cy\*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;  
  where sx,sy,x,y = 0, dx = -0.95, dy = 0.066  
  This is a tent filter, need to look it up.

To begin with, let’s just fire rays through the center of each pixel.

## Milestones

It’s probably a good idea to set some milestones that we want to hit to keep us on track. Can base the milestones off of the checkpoints we had while developing the CG-2 ray tracer as well as those depicted in the GPU path tracer blog. Pbrt chapters could also serve as a good guideline.

1. Architecture and blender integration: Think about how the renderer should be laid out. Also, found an interesting [project](https://agraphicsguy.wordpress.com/2015/09/06/my-external-renderer-for-blender/) wherein someone integrated their path tracer with blender. Might be good to find a way to hook this up from the get-go so we have a direct link to the modeling tool.
2. Object intersection testing: Most basic step that needs to be performed. Can start with the Whitted scene and then move onto more complex polygonal objects.
3. Acceleration structures: Need to look into implementing acceleration structures to speed up intersections with more complex shapes.
4. Moveable camera: Work on implementing a moveable camera. Should be able to pretty much lift the DXCamera implementation.
5. Lighting: The meat of the implementation. Pbrt’s chapters dedicated to the topic are as follows:
   1. Colour and radiometry
   2. Sampling and reconstruction
   3. Reflection models
   4. Materials
   5. Volume scattering
   6. Light sources
   7. Monte Carlo sampling
   8. Light transport: surface reflection
   9. Light transport: volume rendering
   10. Light transport: precomputed light transport

Smallpt’s implementation should be a good start to get us up and running. We can focus on each of the aforementioned topics once that’s done with.

1. Additional effects: Whatever we can think of:
   1. Motion blur
   2. Depth of field
   3. <add more effects here>

### Architecture and blender integration

To begin with, let’s extend the DXSample class from the D3D12 samples to form the basis of the rendering engine. We can then walk pbrt’s simple renderer to get some ideas about how everything should fit together. Also makes sense to look at GPUPathTracer’s architecture as that would most likely be something we would end up finally mimicking.

### Object intersection testing

### Acceleration structures

### Moveable camera

### Lighting

### Additional effects