# 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.