# 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.

### Resource binding

(TODO: Talk about different resource binding methods here.)

Learned something interesting about root descriptors. They're limited to only CBVs and raw/structured buffer UAVs and have the additional limitation that only the GPU virtual address is made available to the hardware. It seems that this is a more efficient binding method than descriptor tables which is probably why it is restricted to the resources which will change most frequently during a frame.

Static samplers seem interesting, they can be baked directly into the shader thus avoiding the need to create descriptors and bind them to the pipeline.

The overall recommendations seems to be as follows:

* Be mindful of the total size available for root arguments (64 DWORDs).
* Organize root arguments in descending order of usage frequency (highest to lowest). An exception would be arguments that are accessed infrequently but require low access latency.
* Give preference to binding constant buffer arguments via root constants and root descriptors as they tend to be accessed more frequently than other resources and can benefit from the reduced latency offered by these two binding methods.
* UAVs have an upper limit to the no. of binds **across all shader stages** depending on the hardware tier we're working with.
* Changing root signatures causes all currently bound root arguments to become undefined.

(TODO: Decide whether to talk about bundles or not.)

#### Root signature at the Pipeline State Object (PSO) as well as at the API level

A bit of confusion can arise from the fact that root signatures can be set from both the API level via Set(Graphics/Compute)RootSignature as well as via PSOs wherein the root signature is one of the parameters to the PSO descriptor. The rationale is seemingly as follows:

* The PSO uses the root signature to generate shader code that knows where root arguments will be located.
* Setting the root signature via the APIs allows command lists to be aware so that APIs like Set(Graphics/Compute)RootDescriptorTable will know where the arguments need to go.
* Setting the root signature via the APIs also allows root parameters which need to be set regardless of which PSO is bound ahead of time.
* Every time a PSO is set, it's root signature is also implicitly and this can lead to subtle issues as it becomes possible to lose track of what the currently set root signature is. Another side effect of this is that root arguments get invalidated whenever the root signature changes. The end result is that if the application is not careful, it is possible to end up with a mismatch between the currently set root signature and root arguments.

Couple of notes regarding setting of a root signature via a PSO:

* PSOs can be implicitly set by the comprising shaders. The root signature becomes invalid, however, if different shaders in the same PSO have different root signatures (the createPSO call fails).
* Setting the root signature parameter explicitly overrides the PSOs inherited from the shaders.
* It is illegal to create a PSO without a root signature.

#### Editing descriptor heaps

* Descriptor heaps can only be created/copied on the CPU. Recording these operations on a command list is not supported. The reasoning apparently is that supporting descriptor heap edits on the CPU as well as the GPU would lead to substantial system complexity. One workaround is to add a constant buffer update to a command list that selects a descriptor out of a descriptor array which needs to be used in a shader.
* Non shader visible descriptor heaps have read/write access by the CPU.
* Shader visible descriptor heaps only have CPU write and GPU read access to optimize data flow. These kinds of heaps can only be used as the destination of descriptor heap edits. This kind of optimal descriptor heap memory (write combined) tends to be limited on discrete GPUs.
  + Write combined memory tends to be somewhere around 96 MB on most systems and is shared across **all** processes.
  + A descriptor heap with a million descriptors takes about 32 MB and it wouldn't take long to bleed the write combined memory dry if the app keeps making max size descriptors. The advice is to avoid doing so if possible.
  + The OS would fall back to system memory if the write combined memory is depleted which can be slower.

#### Setting descriptor heaps

* Modifying descriptor heaps while command lists that reference said heaps are executing should be avoided as doing so could result in a race condition.
* Only one CBV/SRV/UAV and one sampler descriptor heap can be bound at a time. These heaps will be shared between graphics and compute PSOs.
* Setting a descriptor heap can cause a GPU stall as the GPU will wait for any executing command lists to complete before performing the set operation. One way to hide the stall is to do the set adjacent to any other operation that will already cause a stall like at the beginning of a command list.
* Bundles can only call SetDescriptorHeaps once and the heaps it sets must match those set by the parent command list.

#### Descriptor heap arrangement strategies

* Have separate sets of descriptors per draw. This is a simple approach but can lead to duplication and can eat up descriptor heap space quickly. Multiple descriptor heaps can be used so that the CPU can write to some of them while the GPU can read from the others.
* Pre-fill the descriptor heap with descriptors based on some categorization such as object ID or material ID and then just set descriptor tables accordingly.
* Use root constants and/or root descriptors for constants which change frequently as there is less indirection involved as compared to using root descriptor tables.
* Little bit of a restriction with tier 1/2 resource binding H/W wherein descriptors in a descriptor table which are referenced by a shader need to be populated before the shader begins execution. This needs to be done irrespective of whether the shader actually ends up using it or not due to branching. Null descriptors can be used to fill the gaps if necessary. This applies to the following descriptor types per H/W tier:
  + Tier 1: CBV, SRV, UAV and sampler.
  + Tier 2: CBVs and UAVs
  + Tier 3: N/A

#### Null descriptors

* Null descriptors, when referenced, result in default behaviour as if the pipeline referenced a bind slot with nothing bound.
  + Reads return 0 for components within/without the format name. An exception is alpha which returns 1.0f/0x1 depending on the format name.
  + UAV writes are dropped.
* Null descriptors are created by setting the pResource parameter in Create\*View to NULL.
* One thing to remember is that the view description should still have a valid configuration with respect to the shader that it is going to be used with. The reason is that some H/W can crash if a shader references a descriptor with an incompatible view description. Root descriptors are an exception as they essentially contain just a GPU virtual address. Accessing an invalid descriptor would result in a page fault.
* One side note is that creating a view by passing in a resource but no view description results in a view that describes the resource as strictly as possible.
  + The format needs to be strongly typed as opposed to one of the typeless formats.

#### Root signature switching

* Not recommended to switch root signatures every PSO change as that can be inefficient.
* It's possible that multiple PSOs can share the same root signature which can save on the overall no. of root signatures required.

#### Graphics and Compute roots

* The graphics and compute pipelines in D3D12 have distinct root behaviours that are mutually exclusive.
* The current PSO decides whether the graphics or compute root signature/arguments is used.

#### Root signature authoring

There are two ways to write root signatures:

1. Programmatically in C++: The root signatures will be generated at runtime.
2. Offline in HLSL: The root signature can be declared as a string separate from the rest of the HLSL. Commonly used root signatures can be #include-d in all shaders that use them. The root signature can also be added offline using fxc.

#### HLSL root signature

Couple of interesting features of the HLSL root signature:

* If multiple root arguments in a root signature contains bindings that reference the same bind slot, a parameter called 'space=' can be defined along with the first of the root parameters in question to avoid said conflicts.
* If a descriptor table has multiple descriptors of a given type, the parameter 'numDescriptors' can be defined to indicate how many descriptors are present. The parameter can have the value 'unbounded' to indicate that the no. of descriptors of that particular type will be unbounded. This can only be set to the last of the descriptor types at the end of the descriptor table.
* It is possible to alias multiple bind points in that they can refer to the same descriptor. Aliasing descriptors can be specified using the 'offset' parameter.

#### Basic architecture

* We have the MiniEngine framework to abstract away a lot of the boilerplate code
* The DX11 ray tracer should give us a good idea of how the architecture should be.

### Object intersection testing

To start with, let’s see if we can compute ray directions from the thread ID of each pixel.

Looks like there are a couple of camera specific transformations to be kept in mind:

* Camera to world space
* Camera to projection space
* Projection to raster space. Composed of 3 steps:
  + Translate origin (center point) to top left corner of the screen.
  + Scale by reciprocal of screen width and height to transform coordinates from signed normalized [-1,1] to unsigned normalized [0,1].
  + Scale by window width and height to get range in [0,width-1], [0,height-1].
* Raster to projection space
* Raster to camera space

The current pixel (x,y) location is transformed to camera space by the RasterToCamera transform.

Pbrt transforms the ray from camera to world to object space to perform intersection testing.

Really need a sandbox project to test out these transforms.

### Moveable camera

We can adapt the DXCamera class to fit our needs. The view and proj matrix implementations would differ based on the path tracer’s requirements.

That was easier than expected ☺ The camera does tend to fall apart once you start viewing it at odd angles though, try to see why that is.

Whoops, looks like we forgot to pass in the camera params to determine the ray origin and dir :D

### Acceleration structures

### Color representation

### Sampling and reconstruction

### Reflection and materials

### Volume scattering

### Monte Carlo Integration

### Light transport

### Additional effects

## MiniEngine evaluation

Noticed that it has this interesting way of compiling shaders. Shaders are compiled into C++ header files and a global byte array is created which can be directly used in the root signature definition. The required compiled shader headers are included just like any other headers. Definitely saves us the need of creating a blob at runtime and loading it into the program.

The root signature object is inited by calling the Reset method specifying the no. of bind slots to be used as well as the no. of static samplers. The [] operator is overloaded to access each bind slot’s root parameter which can be initialized using a variety of Init methods. An InitStaticSampler method is available to initialize static samplers.

PSO objects have set methods to initialize state. There is also a Finalize method that needs to be called once all the state has been set.

A model class is available which can prove convenient once we’re dealing with more advanced shapes. It also has support for pbrt meshes ☺

A fully functional camera class is available. There is also a separate CameraController which purportedly is used to tweak the motion parameters of the camera.

There are GraphicsContext and ComputeContext classes to represent the graphics and compute pipelines which encapsulate the corresponding state setting and draw/dispatch functionality.

It seems to have a single global descriptor heap encapsulated in a class called DescriptorAllocator. This class is used internally by resource wrapper classes like ColorBuffer, DepthBuffer to store resource views. It looks like for every resource, RTV, SRV and UAVs are automatically created.

The refactor will have to go like this:

rsmgptEngine.h:

* rsmgptEngine can extend the IGameApp class. The pure virtual functions of DXSample will be replaced by those of IGameApp.
  + On second thought, the GameCore::RunApplication method seems to do a lot of extra work on top of what we require. Might be worthwhile to implement device creation and the swap chain ourselves as we don’t need a lot of the extra stuff.
* m\_viewport and m\_scissorRect can stay.
* m\_commandQueue will become a GraphicsContext instance and m\_computeCommandQueue will become a ComputeContext instance. Their usage will change however as they will no longer be queue objects but rather, rendering contexts.
  + Actually, it seems we can’t have instances of these. References to them are returned by their corresponding Begin methods.
* m\_rootSignature and m\_computeRootSignature will become RootSignature types.
* m\_pipelineState will become a GraphicsPSO instance and m\_computePipelineState will become a ComputePSO instance.
* m\_vertexBuffer and m\_constantBuffer will become GpuBuffer instances but let’s see if we can do without m\_vertexBuffer by using the full-screen triangle method that doesn’t require a vertex buffer.
* m\_depthStencil will become a DepthBuffer instance.
* m\_pathTracerOutput will become a ColorBuffer instance.
* Include the following headers:
  + GameCore.h
  + GraphicsCore.h
  + CommandContext.h
  + SamplerManager.h

rsmgptEngine.cpp

* Ctor
  + No changes required here.
* Startup
  + All the device, swap chain, command queue, command allocator, descriptor heap init code can be skipped.
    - If we’re not extending IGameApp, this will have to be reinstated. The render targets can be redefined as ColorBuffer instances.
  + Update m\_rootSignature and m\_computeRootSignature initialization as per RootSignature’s implementation.
  + Update shader loading to include the compiled shader header files. Shaders will have to be updated to compile to .h instead of .cso.
  + Update m\_pipelineState and m\_computePipelineState initialization.
  + Update vertex buffer initialization, may have to use the gfx context to make the copy happen.
  + Update m\_depthStencil, m\_constantBuffer and m\_pathTracerOutput initialization.
  + Synchronization code may no longer be required here.
* Cleanup
  + Synchronization code may no longer be required here.
* Update
  + Nothing for now.
* RenderScene
  + No command allocators/lists to reset here.
  + Setup the compute context and dispatch.
  + Setup the graphics context.
  + Transition the render target from present to render target and clear.
  + Transition the path tracer output from UAV to SRV and draw the full-screen triangle.
  + Transition the path tracer output from SRV to UAV and the render target from render target to present.
  + Close and execute the gfx context.

Everything looks highly interconnected, it might not be that easy to use only part of the library after all ☹

The full screen triangle SV\_VertexID trick doesn’t seem to be working for some reason. Can probably shelve it for now but might be nifty to get it working at some point.

There were some interesting takeaways from the porting exercise though. There are some aspects of miniengine that are definitely worth adopting:

* DescriptorHeap:
  + Each instance can own a specific type of descriptor heap and populates it with a predefined no. of descriptors.
  + Every time a view is created, the allocate function is called which allocates the given no. of descriptors and returns a handle corresponding to the first view of the set.
  + It might be useful to have add(SRV/RTV/UAV) methods which can be used to create views corresponding to a resource and then store off the handle in the resource itself.
  + Might be useful to have a map internally which can be used to tie CPU/GPU handles to some sort of identifier (say a string).
* RootSignature
  + Manager class for handling root signatures.
  + Thin wrapper to encapsulate a lot of the boilerplate code.
  + Param ctor accepts no. of root parameters and static samplers.
  + [] operator is used to index into the root parameters.
  + InitStaticSampler is used to init static samplers.
  + Finalize method is used to serialize and finally create the root signature.
  + MiniEngine implements a descriptor cache in the RootSignature class. Can skip this for now and revisit it if deemed necessary.
* RootParameter
  + Manager class for handling root parameters.
  + Thin wrapper to encapsulate a lot of the boilerplate code.
  + InitAsConstantBufferView and InitAsDescriptorTable can fall through to their CD3DX12\_ROOT\_PARAMETER counterparts.
  + InitAsDescriptorRange can create a table of a single entry.
  + In the end, decided against this class as InitDescriptorRange required explicitly allocating and deallocating the descriptor ranges which would not have allowed the reuse of CD3DX12\_ROOT\_PARAMETER as it doesn’t own the memory of it’s descriptor ranges.
* Texture:
  + Resource class which holds 2d textures (for now).
  + Creates the resource and also has the optional functionality to create SRV, RTV, UAV.
  + Views are created using the aforementioned allocator’s interface methods. It will own the CPU/GPU handles that are returned by the interface.
* GpuBuffer:
  + Resource class for constant/vertex/index buffers.
  + Accepts a pointer to the initial data and has methods to create constant/vertex/index buffer views.

## DX11 ray tracer evaluation

The gtkmm dependency is a little bit of a pain with all the distributed headers. Luckily, it seems there are property sheets that we can use.

TODO: Continue this evaluation.

## Features wish-list breakdown

1. Basic infrastructure
   1. We have the MiniEngine framework to abstract away a lot of the boilerplate code
   2. The DX11 ray tracer should give us a good idea of how the architecture should be.
2. Object intersections
3. Interactive camera
4. Acceleration structures
5. Color representation
6. Sampling and reconstruction
7. Reflection and materials
8. Volume scattering
9. Monte Carlo Integration
10. Light transport