TUTORIAL
OPTIX 7, STEP BY STEP

Ingo Wald

MOTIVATION

WHY THIS TUTORIAL?

At this point, you now know everything about Contexts, Modules, Pipelines, Shader Programs, Hitgroup Records, SBTs, SBT instance offsets, SBT indexing per ray type, AS'es, AS built inputs, … and can build some amazing Optix7 applications with that!

Just in case: going to walk you through some of that, step by step…

MOTIVATION

WHY THIS TUTORIAL?

Tutorial motivated by Chris Wyman's 2018 "Intro to DXR" Tutorial

- http://intro-to-dxr.cwyman.org/
- If you haven't seen it, go check it out, it's *really* good!
  (and totally complementary to this course)

Only "problem" with this tutorial: Ignored the "setup" part …

- Assumed SBTs and AS'es already set up, then focused on shading side
- Great if you have tool that does that (→Falcor), but what if not?

→Goal of this tutorial: Walk novice user through all the steps to get started

TUTORIAL: OPTIX 7 IN 10 STEPS

GO PLAY WITH IT!

All code for these examples is available online

- https://www.gitlab.com/ingowald/optix7course

Builds on both Windows and Linux (tested Ubuntu 18&19), few dependencies

- Windows: need Visual Studio, Cmake, CUDA 10.1, and OptiX 7 SDK/driver
- Linux: cmake, libglfw3-dev, CUDA 10.1, and Optix 7 SDK/driver

Data: download “crytec sponza” model from https://casual-effects.com/data/

- … then unpack zip file into project folder (hardcoded paths in samples)

BEFORE I BEGIN …

(A WORD OF CAUTION)

First: Optix 7 is *amazing!*

→ *Gives you back control over what happens when, how.*

BUT - heads-up: Optix 7 is *much* more “explicit” than Optix <= 6.

a) “Somewhat” more setup code…

b) It’s *your* job to set it all up

→ First examples will be the “hardest” / most detailed … bear with me.

EXAMPLE 1: HELLO OPTIX

CONSOLE “HELLO OPTIX” PROGRAM (SANITY CHECKING)

Goal: Trivially simple “Hello World”

- Initialize Optix (optixInit()), and error-check
- Say hello, and exit.

→Primarily “sanity check” that system can compile, link, and run Optix 7 apps

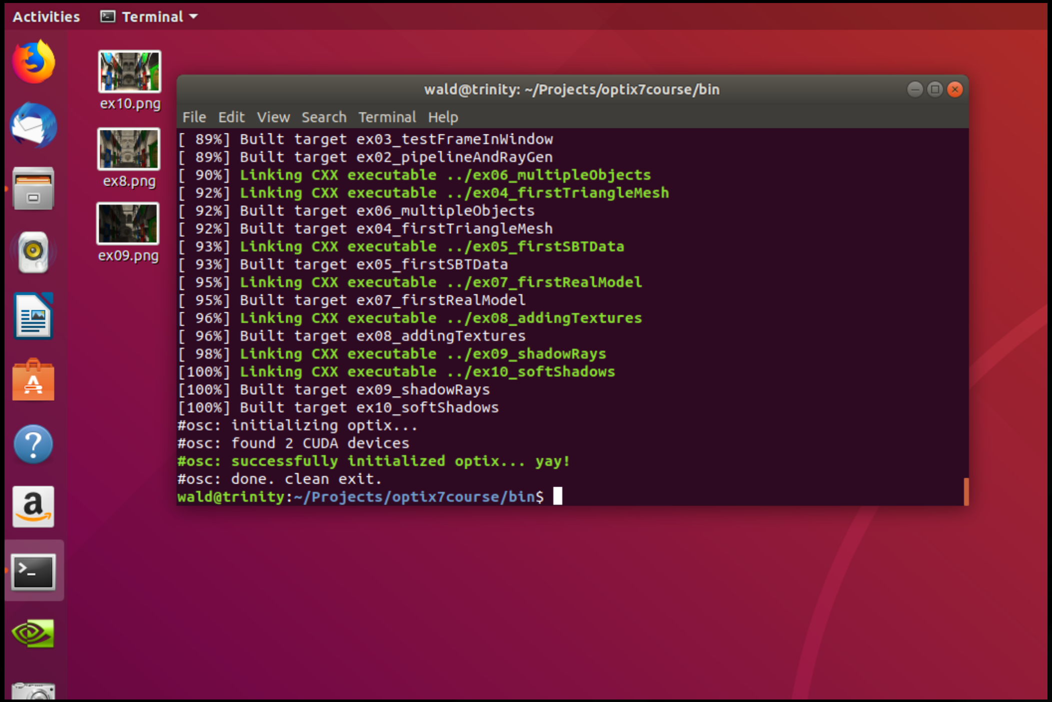
- Building the code: cmake should find paths automatically
- Running: Unlike Optix <= 6, setting LD_LIBRARY_PATH (Linux) respectively PATH (Windows system environment) should no longer be required

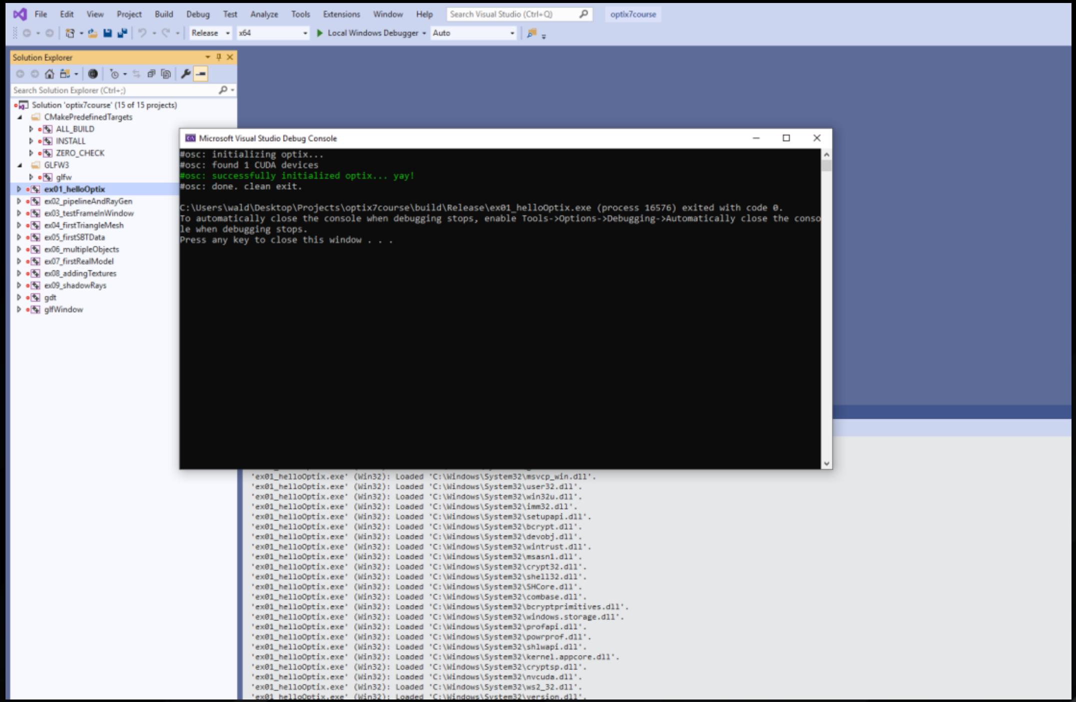
EXAMPLE 1: HELLO OPTIX

CONSOLE “HELLO OPTIX” PROGRAM (SANITY CHECKING)

Works in both Linux …

… and in Windows





EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLETE OPTIX PIPELINE SETUP AND RAYGEN LAUNCH

Goal: First “real” optix launch that computes “some” color value per pixel

- Need a frame buffer, trivial raygen program (compute color), and optixLaunch()

EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLETE OPTIX PIPELINE SETUP AND RAYGEN LAUNCH

Goal: First “real” optix launch that computes “some” color value per pixel

- Need a frame buffer, trivial raygen program (compute color), and optixLaunch()

Well – it’s a little bit more complicated than that …

- Also need way to pass frame buffer to raygen (→launchParams)

EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLETE OPTIX PIPELINE SETUP AND RAYGEN LAUNCH

Goal: First “real” optix launch that computes “some” color value per pixel

- Need a frame buffer, trivial raygen program (compute color), and optixLaunch()

Well – it’s a little bit more complicated than that …

- Also need way to pass frame buffer to raygen (→launchParams)
- Also need way to get raygen program “into” optix: compilation of CUDA file, embedding of PTX code, creation of context, module, pipeline,…

EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLETE OPTIX PIPELINE SETUP AND RAYGEN LAUNCH

Goal: First "real" optix launch that computes "some" color value per pixel

- Need a frame buffer, trivial raygen program (compute color), and optixLaunch()

Well – it's a little bit more complicated than that …

- Also need way to pass frame buffer to raygen (→launchParams)
- Also need way to get raygen program "into" optix: compilation of CUDA file, embedding of PTX code, creation of context, module, pipeline,…
- Raygen is a "shader" : also need full SBT ('cause raygen is a *shader*), …

→ Heads-up: will be (by far) most detailed example (it'll get simpler after this)

EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLETE OPTIX PIPELINE SETUP AND RAYGEN LAUNCH

High-level view: four big steps (each w/ multiple sub-steps)

1) Create (device-side) raygen program that computes pixel colors
2) Create Optix “pipeline”
   - Think “which *kind* of programs we want the device to run”
3) Create Shader Binding Table (SBT)
   - Think “which *exact configuration* of these programs we want to run”
4) Create a frame buffer, and launch raygen program

EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLETE OPTIX PIPELINE SETUP AND RAYGEN LAUNCH

High-level view: four big steps (each w/ multiple sub-steps)

1) Create (device-side) raygen program that computes pixel colors
2) Create Optix “pipeline”
   - Think “which *kind* of programs we want the device to run”
3) Create Shader Binding Table (SBT)
   - Think “which *exact configuration* of these programs we want to run”
4) Create a frame buffer, and launch raygen program

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM ...

One step back: How this would have looked in Optix 6:

```
// “global” variants as parameters to raygen program
// rtBuffer<> type wraps device buffers
rtBuffer<uint32_t> framebuffer;
int2               fbSize;

// program accesses “global” vars:
RT_PROGRAM void raygen()
{
   framebuffer[rtLaunchIndex()] = …;
}
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

In Optix 7:

```
// only one global: user-supplied LaunchParams struct
extern "C" __constant__ LaunchParams launchParams;

// raygen program:
extern "C" __global__ void __raygen__renderFrame()
{
   launchParams.framebuffer[…] = …;
}
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

In Optix 7:

```
// only one global: user-supplied LaunchParams struct
extern “C” __constant__ LaunchParams launchParams;

// raygen program:
extern “C” __global__ void __raygen__renderFrame()
{
   launchParams.framebuffer[…] = …
}
```

First: mind the naming requirements

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

In Optix 7:

- *one* global (constant) struct (struct type supplied by user)
- In constant memory. will talk later about this gets there…

```
// only one global: user-supplied LaunchParams struct
extern "C" __constant__ LaunchParams launchParams;

// raygen program:
extern "C" __global__ void __raygen__renderFrame()
{
   launchParams.framebuffer[…] = …;
}
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

First, need a “LaunchParams” struct that captures what we want to pass:

```
// LauchParams.h
struct LaunchParams {
   vec2i     fbSize;
   uint32_t *fbPixels;
};
```

Need this struct in both host and device code → put in separate header file

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

First, need a “LaunchParams” struct that captures what we want to pass:

```
// LauchParams.h
struct LaunchParams {
   vec2i     fbSize;
   uint32_t *fbPixels;
};
```

Your **class: can put in there what you want**

Need this struct in both host and device code → put in separate header file

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

First, need a “LaunchParams” struct that captures what we want to pass:

```
// LauchParams.h
struct LaunchParams {
   vec2i     fbSize;
   uint32_t *fbPixels;
};
```

No more “rtBuffer<>”: plain C++/CUDA pointers (*your* job to do cudaMalloc, cudaMemcpy…)

Need this struct in both host and device code → put in separate header file

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

Next: Write out device code in separate CUDA file (eg, “deviceCode.cu”)

```
// launch params:
extern “C” __constant__ LaunchParams launchParams;

// raygen program:
extern “C” __global__ void __raygen__renderFrame()
{
   launchParams.framebuffer[…] = …;
}

// dummy miss and hit programs:
extern “C” __global__ void __miss__radiance() { /* empty */ }
extern “C” __global__ void __closesthit__radiance() { /* empty */ }
…
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM

The actual launch params and program we need…

Next: Write out device code in separate CUDA file (eg, "deviceCode.cu")

```
// launch params:
extern "C" __constant__ LaunchParams launchParams;

// raygen program:
extern "C" __global__ void __raygen__renderFrame()
{
   launchParams.framebuffer[…] = …;
}

// dummy miss and hit programs:
extern "C" __global__ void __miss__radiance() { /* empty */ }
extern "C" __global__ void __closesthit__radiance() { /* empty */ }
…
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

Next: Write out device code in separate CUDA file (eg, “deviceCode.cu”)

```
// launch params:
extern “C” __constant__ LaunchParams launchParams;

// raygen program:
extern “C” __global__ void __raygen__renderFrame()
{
    launchParams.framebuffer[…] = .
}

// dummy miss and hit programs:
extern “C” __global__ void __miss__radiance() { /* empty */ }
extern “C” __global__ void __closesthit__radiance() { /* empty */ }
```

“dummy” miss and hit programs to make SBT happy
(will flesh those out later on)
optixLaunch assumes error if SBT has missing data

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

Next: add Cmake rules to compile .cu file, and embed PTX in binary

```
// example2/CMakeLists.txt:

include “cmake/configure_optix.cmake”

cuda_compile_and_embed(deviceCode, “deviceCode.cu”)

add_executable(example2
   …
   ${deviceCode})
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

Next: add Cmake rules to compile .cu file, and embed PTX in binary

```
// example2/CMakeLists.txt:

include “cmake/configure_optix.cmake”

cuda_compile_and_embed(

add_executable(example2
   …
   ${deviceCode})
```

Some cmake helper scripts I added to samples…
Feel free to use, or use your own.

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

Next: add Cmake rules to compile .cu file, and embed PTX in binary

```
// example2/CMakeLists.txt:

include “cmake/configure_optix.cmake”

cuda_compile_and_embed(deviceCode, “deviceCode.cu”)

add_executable(example2
   …
   ${deviceCode})
```

Provided helper rule that:

a) invokes CUDA compiler to compile devicecode
b) embeds generated PTX code in a global string

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART I : WE NEED A (DEVICE-SIDE) RAYGEN PROGRAM …

Next: add Cmake rules to compile .cu file, and embed PTX in binary

```
// example2/CMakeLists.txt:

include “cmake/configure_optix.cmake”

cuda_compile_and_embed(deviceCode, “deviceCode.cu”)

add_executable(example2
   …
   ${deviceCode})
```

Add embedded device code to binary

EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLEX OPTIX PIPELINE AND RAYGEN LAUNCH

High-level view: four big steps (each w/ multiple sub-steps)

1) Create (device-side) raygen program that computes pixel colors
2) Create Optix “pipeline”
   - Think “which *kind* of programs we want the device to run”
3) Create Shader Binding Table (SBT)
   - Think “which *exact configuration* of these programs we want to run”
4) Create a frame buffer, and launch raygen program

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART II : SET UP THE PIPELINE (NO ACCEL YET IN THIS EXAMPLE)

1) Initialize Optix

```
// example2/SampleRenderer.cpp:
SampleRenderer::initOptix()
{
    …
    OPTIX_CHECK( optixInit() );
    …
}
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART II : SET UP THE PIPELINE (NO ACCEL YET IN THIS EXAMPLE)

1) Initialize Optix
2) Create Optix Context

```
// example2/SampleRenderer.cpp:
SampleRenderer::createContext()
{
   …
   cudaSetDevice(…);               // set device we want to run on
   cudaStreamCreate(…);            // create a stream (for later)
   cuCtxGetCurrent(…);             // get current CUDA device context
   optixDeviceContextCreate(…);    // create optix context
   …
}
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART II : SET UP THE PIPELINE (NO ACCEL YET IN THIS EXAMPLE)

1) Initialize Optix
2) Create Optix Context
3) Create Optix Module

```
// the embedded PTX string (see prev slides)
extern “C” char deviceCode[];

void SampleRenderer::createModule() {
   …
   compileOptions = /* see code */
   linkOptions    = /* … */
   optixModuleCreateFromPTX(compileOptions,linkOptions,deviceCode,…);
   …
}
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART II : SET UP THE PIPELINE (NO ACCEL YET IN THIS EXAMPLE)

1) Initialize Optix
2) Create Optix Context
3) Create Optix Module

```
// the embed
extern “C” c

void SampleR        .createModule() {
   …
   compileOptions = /* see code */
   linkOptions    = /* … */
   optixModuleCreateFromPTX(compileOptions,linkOptions,deviceCode,…);
   …
}
```

Set pipeline configuration flags: max hierarchy depth, register count, max num attributes, name of launch params variable, etc (no details here, see sample code)

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART II : SET UP THE PIPELINE (NO ACCEL YET IN THIS EXAMPLE)

1) Initialize Optix
2) Create Optix Context
3) Create Optix Module

```
// the embedded PTX string (see prev slides)
extern "C" char deviceCode[];

void SampleRenderer::createModule() {
   …
   compileOptions = /* see code */
   linkOptions    = /* … */
   optixModuleCreateFromPTX(compileOptions,linkOptions,deviceCode,…);
   …
}
```

Create module from embedded PTX code

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART II : SET UP THE PIPELINE (NO ACCEL YET IN THIS EXAMPLE)

1) Initialize Optix
2) Create Optix Context
3) Create Optix Module
4) Set up required *ProgramGroups* (raygen, miss, hitgroup) that go *into* pipeline

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART II : SET UP THE PIPELINE (NO ACCEL YET IN THIS EXAMPLE)

4) Set up *ProgramGroups*: first, *raygen* PG (we need only one)

```
// set up program group specification:
raygenPGs.resize(1);

// set up program group specification:
OptixProgramGroupOptions options = {};
OptixProgramGroupDesc desc = {};
desc.kind = OPTIX_PROGRAM_GROUP_RAYGEN;
desc.raygen.module = module;
desc.raygen.entryPointFunctionName = “__raygen__renderFrame”;

// tell optix to create the PG:
optixProgramGroupCreate(context,desc,1,options,&raygenPGs[0]);
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART II : SET UP THE PIPELINE (NO ACCEL YET IN THIS EXAMPLE)

4) Set up *ProgramGroups*: then, *miss* and *hitgroup* program groups (one each)

- Same as for raygen
- One entry each, fill with our dummy functions

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART II : SET UP THE PIPELINE (NO ACCEL YET IN THIS EXAMPLE)

5) Create Pipeline

```
// create list of all PGs to go into this pipeline:
std::vector<OptixProgramGroup> allPGs = raygenPGs + missPGs + hitgroupPGs;

// create pipeline
optixPipelineCreate(context,&compileOptions,&linkOptions,… allPGs …);
```

…. Yay! We have a pipeline!

EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLEX OPTIX PIPELINE AND RAYGEN LAUNCH

High-level view: four big steps (each w/ multiple sub-steps)

1) Create (device-side) raygen program that computes pixel colors
2) Create Optix "pipeline"
   - Think "which *kind* of programs we want the device to run"
3) Create Shader Binding Table (SBT)
   - Think "which *exact configuration* of these programs we want to run"
4) Create a frame buffer, and launch raygen program

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART III : CREATE OUR SBT

Shader Binding Table (SBT): list of raygen, miss, and hitgroup “records” to run

- Each record contains header (describes PG to run), and user-supplied data
- E.g., a simple raygen SBT record:

```
// our Raygen SBT record (alignment is required)
struct __align__(OPTIX_SBT_RECORD_ALIGNMENT) RaygenRecord
{
   // must have this header (encodes which PG to run)
   char header[OPTIX_SBT_HEADER_SIZE];
   // user-supplied data we want to pass to this program instance
   // (none, in this simple example)
   char ourUserData[0];
};
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART III : CREATE OUR SBT

1) create all the raygen programs we want to use (exactly one …)

```
// create and fill in raygen records
std::vector<RaygenRecord> raygenRecords(1);
// specify the type of program this record is to run:
optixSbtRecordPackHeader(raygenPG[0],&raygenRecords[0]);
// fill in the user data:
raygenRecords[0].userData = ... // not in this tiny example

// use CUDA to upload to device
void *d_raygenRecords;
cudaMalloc(…);
cudaMemcpy(…);
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART III : CREATE OUR SBT

2) do same for miss program record(s)

- Usually, need one miss program record per ray type
- Won't actually use any rays in this example, but use 1 to avoid null entry…
- Upload all records into one CUDA buffer

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART III : CREATE OUR SBT

2) do same for miss program record(s)

- Usually, need one miss program record per ray type
- Won't actually use any rays in this example, but use 1 to avoid null entry…
- Upload all records into one CUDA buffer

3) Same for hitgroup records

- Usually one per ray type and object instance (numObjects x numRayTypes)
- As above: don't use any, just create one dummy for now
- Upload all records into one CUDA buffer

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART III : CREATE OUR SBT

4) Create the SBT:

```
// the SBT to fill in
OptixShaderBindingTable sbt = {};
// fill in raygen record
sbt.raygenRecord = d_raygenRecords;
// fill in miss records
sbt.missRecordsBase = d_missRecords;
sbt.missRecordsCount = 1;
sbt.missRecordsStrideInBytes = sizeof(MissRecord);
// fill in hitgroup records
sbt.hitgroupRecordsBase = d_hitgroupRecords;
sbt.hitgroupRecordsCount = 1;
sbt.hitgroupRecordsStrideInBytes = sizeof(HitgroupRecord);
```

→ Yay! We have a valid SBT!!!

EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLEX OPTIX PIPELINE AND RAYGEN LAUNCH

High-level view: four big steps (each w/ multiple sub-steps)

1) Create (device-side) raygen program that computes pixel colors
2) Create Optix “pipeline”
   - Think “which *kind* of programs we want the device to run”
3) Create Shader Binding Table (SBT)
   - Think “which *exact configuration* of these programs we want to run”
4) Create a frame buffer, and launch raygen program

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART IV: READY TO LAUNCH…

1) Create a (CUDA) frame buffer

```
// create the frame buffer
vec2i fbSize(1600,1200);
void *d_framebuffer;
cudaMalloc(&d_framebuffer,…);
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART IV: READY TO LAUNCH…

1) Create a (CUDA) frame buffer

```
// create the frame buffer
vec2i fbSize(1600,1200);
void *d_framebuffer;
cudaMalloc(&d_framebuffer,…);
```

Again: "plain" CudaMalloc here, no special buffer magic (if you want to share pointers across device, NVLink'ed memory, or use host pinned mem … just go for it!)

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART IV: READY TO LAUNCH...

1) Create a (CUDA) frame buffer

2) Fill in and upload launchParams for raygen program to work on

```
// host side copy of launchparams
LaunchParams lp = { fbSize, d_framebuffer };
// upload to CUDA buffer (cudaMalloc, cudaMemcpy)
void *d_launchParams;
cudaMalloc(&d_launchParams,sizeof(LaunchParams);
cudaMemcpy(…);
```

(of course, future launches will only use cudaMemcpy, not cudaMalloc)

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART IV: READY TO LAUNCH...

1) Create a (CUDA) frame buffer
2) Fill in and upload launchParams for raygen program to work on
3) Launch it!

```
// (asynchronously) launch the program:
optixLaunch(// the pipeline we launch:
            pipeline, stream,
            // the launch params to use for the launch
            // (pipeline known name of device-symbol to copy this into!)
            d_launchParams,sizeof(LauchParams),
            // the SBT to launch
            &sbt,
            // dimensions (NxMxK) of this launch
            fbSize.x,fbSize.y,1
            );
```

EXAMPLE 2: FIRST OPTIXLAUNCH()

PART IV: READY TO LAUNCH…

1) Create a (CUDA) frame buffer
2) Fill in and upload launchParams for raygen program to work on
3) Launch it!
4) Mind: launch is async!

→ In this sample, wait for completion

```
// wait for all cuda launches to sync
cudaDeviceSynchronize();
// error checking
If (cudaGetLastError() …);
```

(Of course, a real program might want to do something useful in that time …)

EXAMPLE 2: FIRST OPTIXLAUNCH()

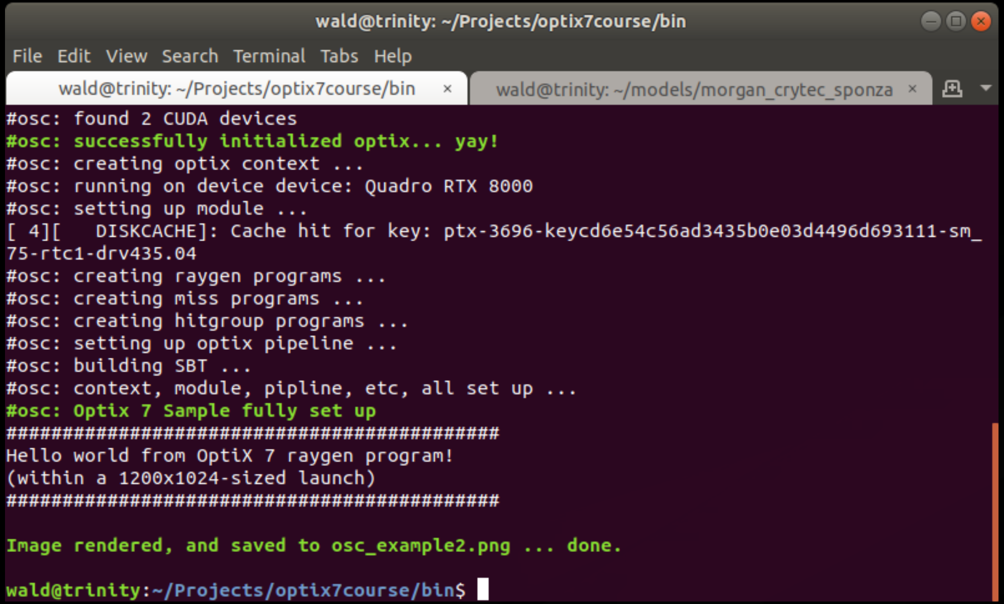
PART IV: READY TO LAUNCH…

1) Create a (CUDA) frame buffer
2) Fill in and upload launchParams for raygen program to work on
3) Launch it!
4) Mind: launch is async!
5) Copy back framebuffer …. And done!

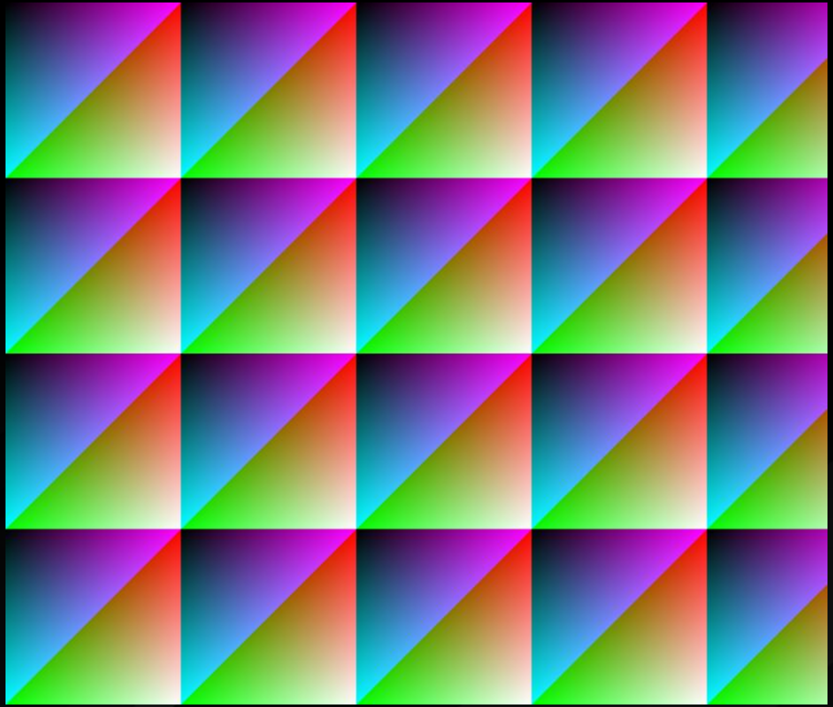
EXAMPLE 2: FIRST OPTIXLAUNCH()

FIRST COMPLEX OPTIX PIPELINE AND RAYGEN LAUNCH

Text console (similar in windows)



Generated PNG file



EXAMPLE 2: FIRST OPTIXLAUNCH()

OK, THAT WAS A LONG ONE …

- That example was (intentionally) very detailed…
  - You *have* to have a context, pipeline, module, and an SBT to do a launch …
  - You have to do it *right*, or you'll get core dumps rather quickly (not too forgiving ☹).
- But: this looks worse than it is…
  - Can largely copy-n-paste from examples (like this, or OptiX 7 SDK Samples)
- From now on, it gets *way* simpler
  - Pretty much only incremental changes and additions to this pipeline

EXAMPLE 3: RENDERING IN A VIEWER

MOVE SAMPLE RENDERER INTO A WINDOW

- OK, now we can render into a file … but offline is boring (?) …

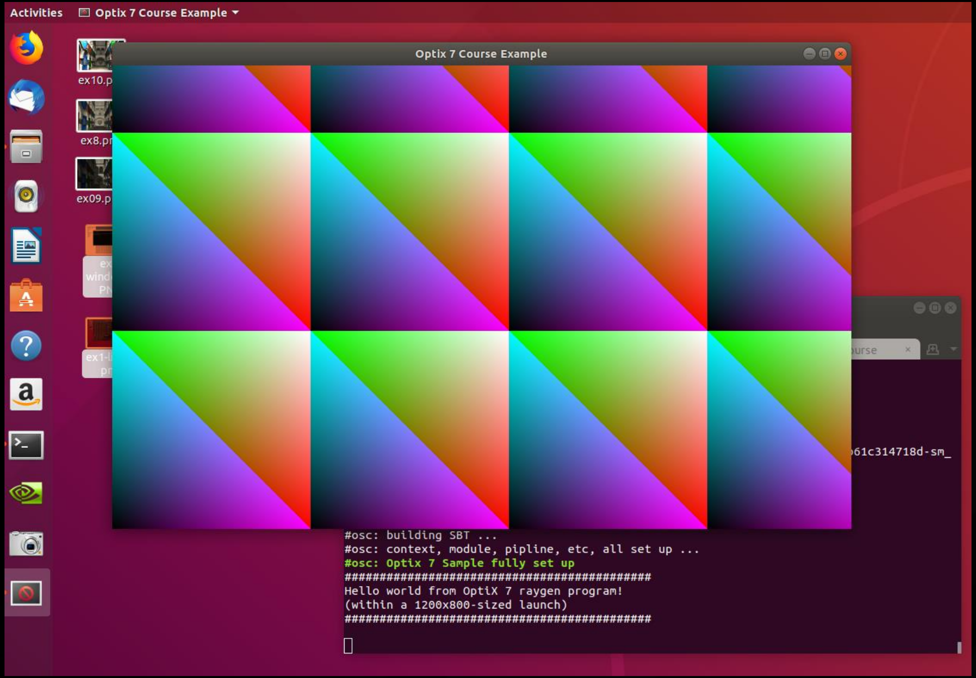
→ Move rendering into a 3D viewer/window based app

  - Use GLFW for windowing – works in windows and linux
  - Re-size framebuffer when window resizes
  - Re-launch raygen every time window wants to redraw
  - That's it … no changes to actual optix code at all

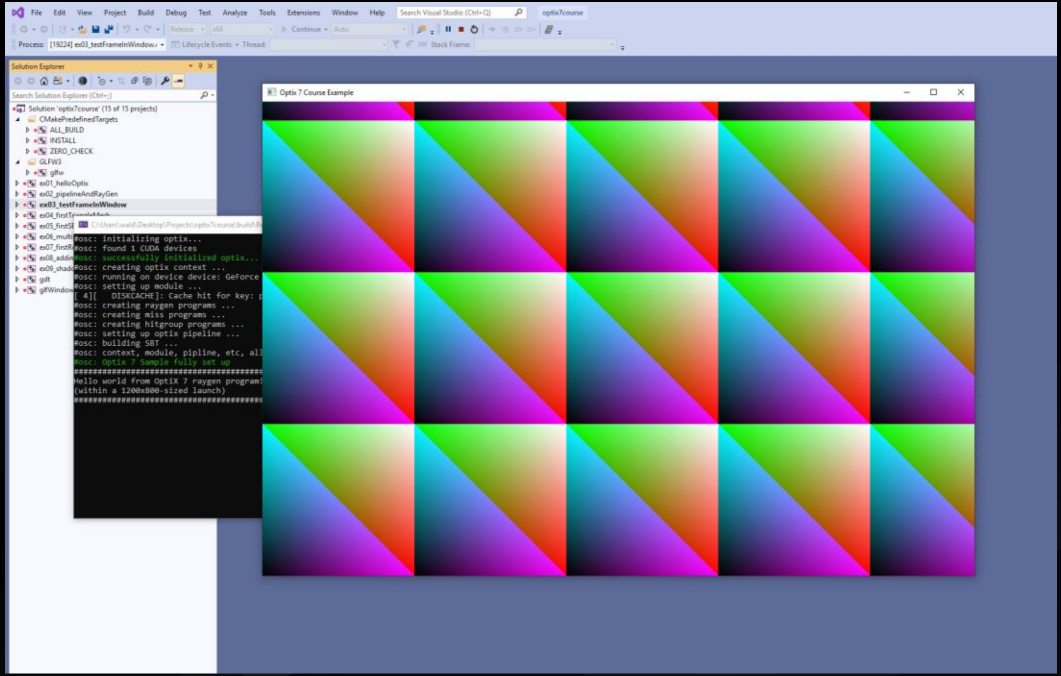
EXAMPLE 3: RENDERING IN A VIEWER

MOVE SAMPLE RENDERER INTO A WINDOW

Again: Works in Linux …



… as well as in Windows



EXAMPLE 4: FIRST TRIANGLE MESH & AS

FIRST ACTUAL TRIANGLE MESH, AS, RAY GENERATION, AND TRACE

- Oh-kay – that was a truly awesome test pattern, but …

→Add some first (triangle) geometry, and trace some rays

  - In this example, will do exactly *one* accel struct, *one* mesh, *one* hitgroup, and *one* ray type

EXAMPLE 4: FIRST TRIANGLE MESH & AS

FIRST ACTUAL TRIANGLE MESH, AS, RAY GENERATION, AND TRACE

On high-level, we need:

- A raygen program that generates real rays, and traces them (device)
- Real hit and miss programs, to process the traced rays (device)
- A means of communicating between programs (per ray data, PRD; on device)
- Some triangles, and an acceleration structure built over it (host code)
- Valid program groups and SBT records for the mesh and miss programs (host)

EXAMPLE 4: FIRST TRIANGLE MESH & AS

DEVICE SIDE: PER RAY DATA

- For this example, use a single vec3f color values
  - Miss programs write background color, hit program writes per-prim pseudocolor
  - Raygen program write this value into frame buffer
- Data we pass with optixTrace is *pointer* to this PRD (so we can modify it)
  - Use helper fcts to encode pointer into two ints (data we pass must be two ints…)

```
// helper that encodes 64-bit pointer into two ints:
void packPointer(T *, int &s0, int &s1);
```

  - … and to decode them pack to a (typed) pointer

```
// helper that encodes from two int PRDs back to (typed) 64-bit pointer
template<typename T> T *getPRD();
```

EXAMPLE 4: FIRST TRIANGLE MESH & AS

DEVICE SIDE: HIT PROGRAM

- Will need an actual hit program (very simple primID shading for now)

```
// first "real" closesthit program
extern "C" __global__ void __closesthit__radiance()
{
   // get (reference to) per-ray data (just a color, in this example)
   vec3f &prd = *getPRD<vec3f>();
   // query ID of primitive we hit
   int primID = optixGetPrimitiveIndex();
   // assign a simple pseudo-color per prim ID
   prd = gdt::randomColor(primID);
}
```

- Anyhit program: leave empty for now

EXAMPLE 4: FIRST TRIANGLE MESH & AS

DEVICE SIDE: MISS PROGRAM

- Similarly, need a miss program for when ray doesn’t hit anything

```
// first “real” miss program
extern “C” __global__ void __closesthit__radiance()
{
   // get (reference to) per-ray data (just a color, in this example)
   vec3f &prd = *getPRD<vec3f>();
   // in this example, just assign constant white background color
   prd = vec3f(1.f);
}
```

EXAMPLE 4: FIRST TRIANGLE MESH & AS

DEVICE SIDE: RAYGEN PROGRAM

- First, add camera data and accel struct handle to LaunchParams

```
// new LaunchParams data, with camera
struct LauchParams
{
   // data from prev example
   struct { … } frame;
   // camera parameters for ray generation, filled in by app
   struct {
      vec3f origin, direction, horizontal, vertical;
   } camera;
   // handle to the accel struct we want to traverse
   OptixTraversableHandle traversable;
};
```

EXAMPLE 4: FIRST TRIANGLE MESH & AS

DEVICE SIDE: RAYGEN PROGRAM

```
// new raygen program
extern “C” __global__ void raygen_renderFrame()
{
   // as per-ray data, use a local color value (could be any other type)
   vec3f prdPixelColor = {};

   // encode (pointer to) this PRD into two ints (can only pass ints as PRD)
   uint32_t u0, u1; // two ints that encode PRD pointer
   packPointer(u0,y1,&prdPixelColor);

   // generate ray, based on pixel ID
   vec2i pixelID = (optixGetLaunchIndex().x,optixGetLauchIndex.y());
   vec3f ray_org = lauchParams.camera.origin;
   vec3f ray_dir = generateRayDirection(pixelID);
   …
```

EXAMPLE 4: FIRST TRIANGLE MESH & AS

DEVICE SIDE: RAYGEN PROGRAM

```
    // new raygen program (cnt'd)
    …
    // trace ray
    optixTrace(// the accel we trace again
               launchParams.traversable,
               // the ray we want to trace
               ray_org,ray_dir,…,
               // the data to index into the SBT
               /*ray type:*/0, /*ray type count:*/1, /*miss type*/0,
               // the actual PRD (encoded pointer to our PRD)
               u0,y1);   // → this will modify prdPixelColor!

    // and write to frame buffer
    launchParams.colorBuffer[…] = prdPixelColor;
};
```

EXAMPLE 4: FIRST TRIANGLE MESH & AS

HOST SIDE: AS GENERATION

- AS generation: set up “TriangleInputs” array (one per different mesh)

```
// for now, hardcode a single mesh
TriangleMesh mesh = createTestMeshData();

// first, upload vertex and index array (using cudaMalloc etc)
vec3f *d_vertex = uploadToCuda(mesh.vertex);
vec3i *d_index  = uploadToCuda(mesh.index);

// set up a single triangle input
OptixBuildInput triangleInput;
triangleInput.type = OPTIX_BUILD_INPUT_TRIANGLES;
triangleInput.triangleArray.vertexBuffers = …;
… etcpp.
```

EXAMPLE 4: FIRST TRIANGLE MESH & AS

HOST SIDE: AS GENERATION

- AS generation: build AS, with compaction (see code for details)

```
// handle to accel we are going to build (as well as device mem for it)
void                      *d_asMemory = nullptr;
OptixTraversableHandle asHandle;
// perform the build
optixAccelComputeMemoryUsage(…);
cudaMalloc(…tempMem…)
optixAccelBuild(…);
cudaMalloc(d_asMemory, compactedSize);
optixAccelCompact(…, asHandle, …);
cudaFree(…);

→ Now have a valid 'asHandle' (as long as asMemory remains valid!)
```

EXAMPLE 4: FIRST TRIANGLE MESH & AS

HOST SIDE: SET UP SBT

For this example, no *real* changes to the SBT:

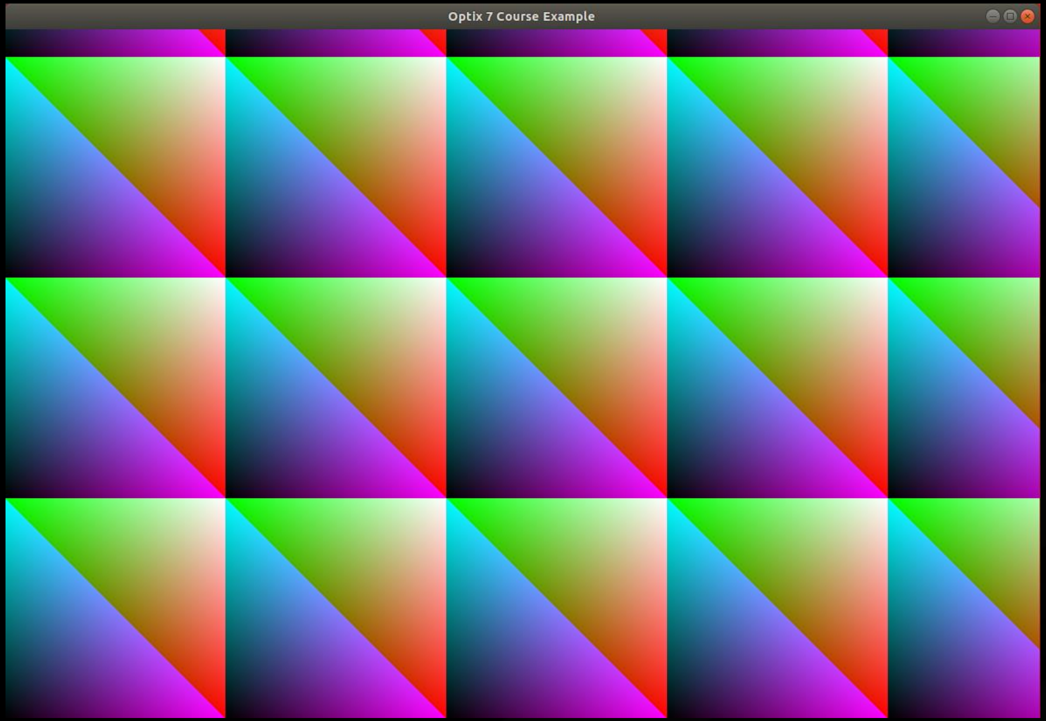
- Still only one ray type and one mesh, so only one hitgroup record
- Still only one miss program
- Still no per-program data (camera and traversable passed through launchparams)

→ Pretty much unchanged (just make sure program groups points to right functions)

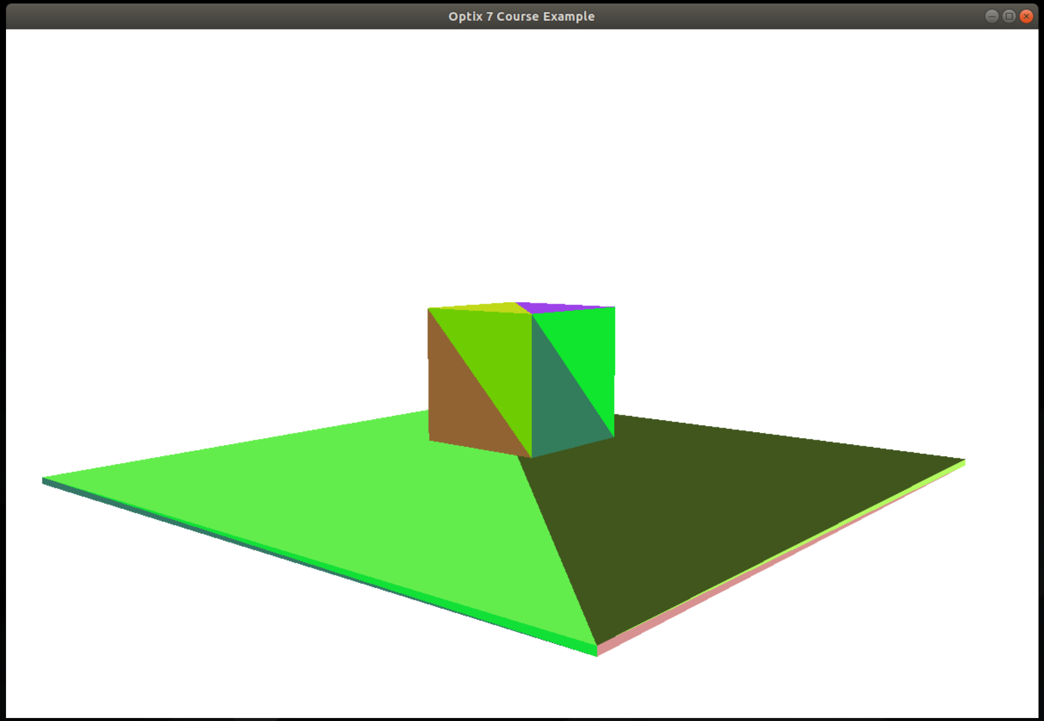
EXAMPLE 4: FIRST TRIANGLE MESH & AS

FIRST ACTUAL TRIANGLE MESH, AS, RAY GENERATION, AND TRACE

Before: test pattern only



Now: w/ (one) “primID” shaded triangle mesh



EXAMPLE 5: FIRST SBT DATA

OH-KAY: NOW LET’S USE THOSE SBT RECORDS TO PASS SOME DATA!

EXAMPLE 5: FIRST SBT DATA

OH-KAY: NOW LET'S USE THOSE SBT RECORDS TO PASS SOME DATA!

1) Extend the Hitgroup record to include some per-mesh data:

```
// our Raygen SBT record (alignment is required)
struct __align__(OPTIX_SBT_RECORD_ALIGNMENT) HitgroupRecord
{
   // must have this header (encodes which PG to run)
   char header[OPTIX_SBT_HEADER_SIZE];
   // our data for this example:
   vec3f *vertexArray;
   vec3i *indexArray;
   vec3f  color;
};
```

EXAMPLE 5: FIRST SBT DATA

OH-KAY: NOW LET'S USE THOSE SBT RECORDS TO PASS SOME DATA!

1) Extend the Hitgroup record to include some per-mesh data

2) Fill in this data in SBT creation

```
// still only one record in this example (one mesh, one ray type)
std::vector<HitGroupRecord> hgRecords(1);
// pack header
optixSbtPackHeader(…);
// store our data
hgRecords[0].vertex = d_vertex; // the one we uploaded during AS creation
hgRecords[0].index  = d_index;  // the one we uploaded during AS creation
hgRecords[0].color  = <pick some color>;
// upload to device
…
```

EXAMPLE 5: FIRST SBT DATA

OH-KAY: NOW LET'S USE THOSE SBT RECORDS TO PASS SOME DATA!

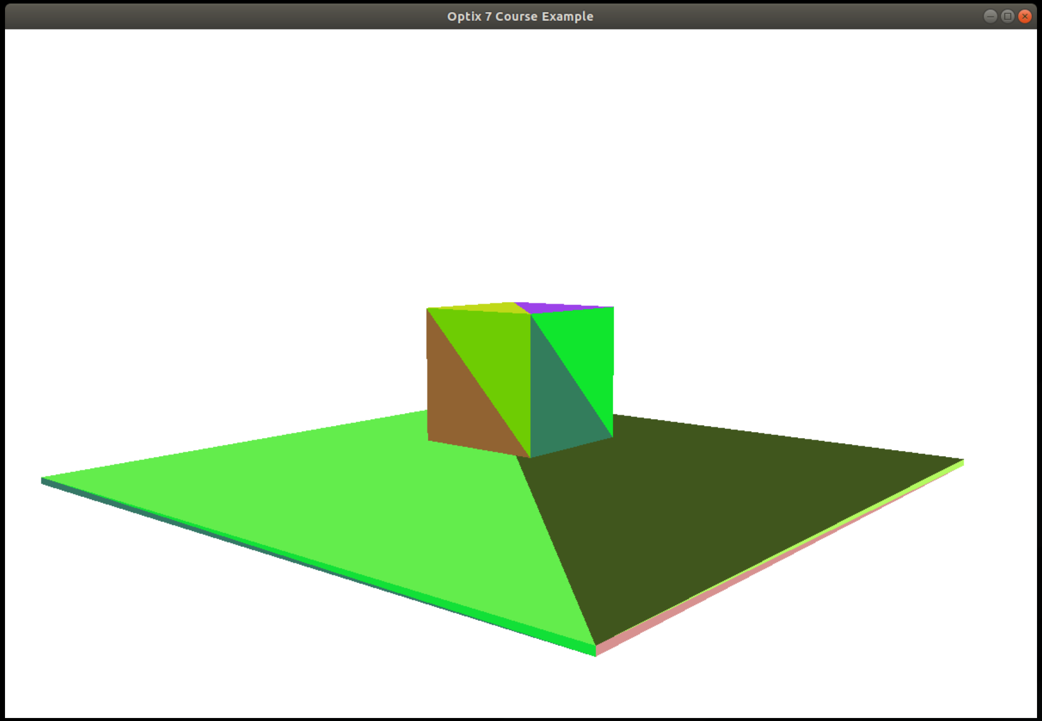
3) Update CH program to use this data

```
// example5/deviceCode.cu
extern "C" __global__ void __closesthit__radiance()
{
   // get SBT data
   HitGroupRecord &mesh = *(HitGroupRecord*)optixGetSbtDataPointer();
   // compute normal
   vec3i index       = mesh.index[optixGetPrimitiveIndex()];
   vec3f normal      = normalize(cross(mesh.vertex[index.y] - …, …));
   // shade with NdotD
   vec3f rayDir      = optixGetWorldRayDirection();
   float cosND       = 0.2f+0.8f*fabsf(normal,rayDir);
   *getPRD<vec3f>() = cosND * mesh.color;
}
```

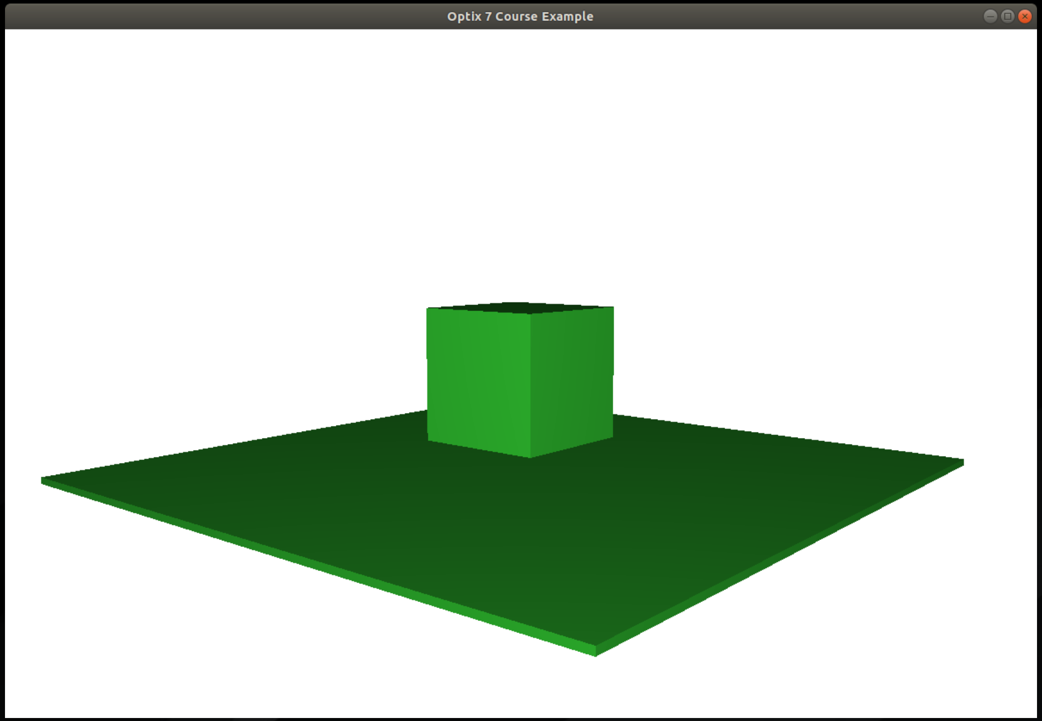
EXAMPLE 5: FIRST SBT DATA

USE SBT RECORD DATA TO PASS PER-MESH USER DATA

Before: no shading data, primID shading only



Now: w/ material color and NdotD shading



EXAMPLE 6: MULTIPLE INDIVIDUAL MESHES

MULTIPLE TRIANGLEINPUTS (IN ACCEL) AND HITGROUP RECORDS (IN SBT)

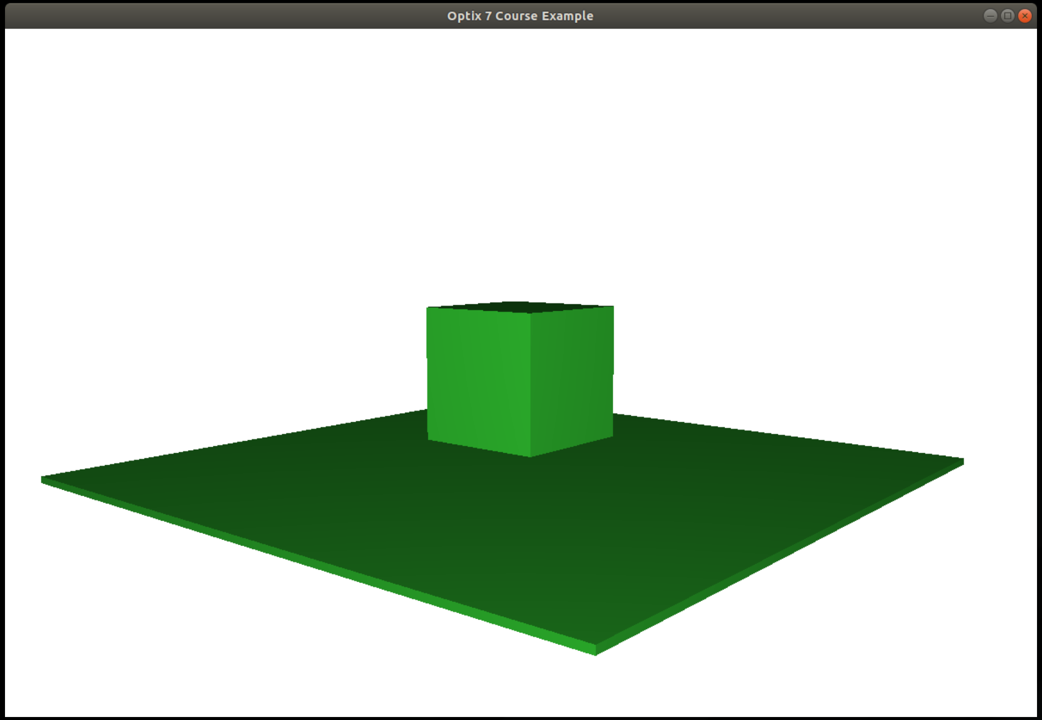
From now on, changes get really, really minimal, with larger outcomes:

- Go from one triangle mesh to 2 meshes (separate mesh for each test cube)
  - Still only one hit program group (both meshes run same *type* of programs) → no change
- Still one accel, but now with 2 build inputs into accel (very few changes)
  - But: each gets one SBT entry → Will now need 2 entries in SBT hitrecords array
- Now 2 entries in SBT hitRecords
  - One ray type, 2 meshes → 2x1=2 entries
- Assign pseudo-color to each mesh …. Aaaand ….

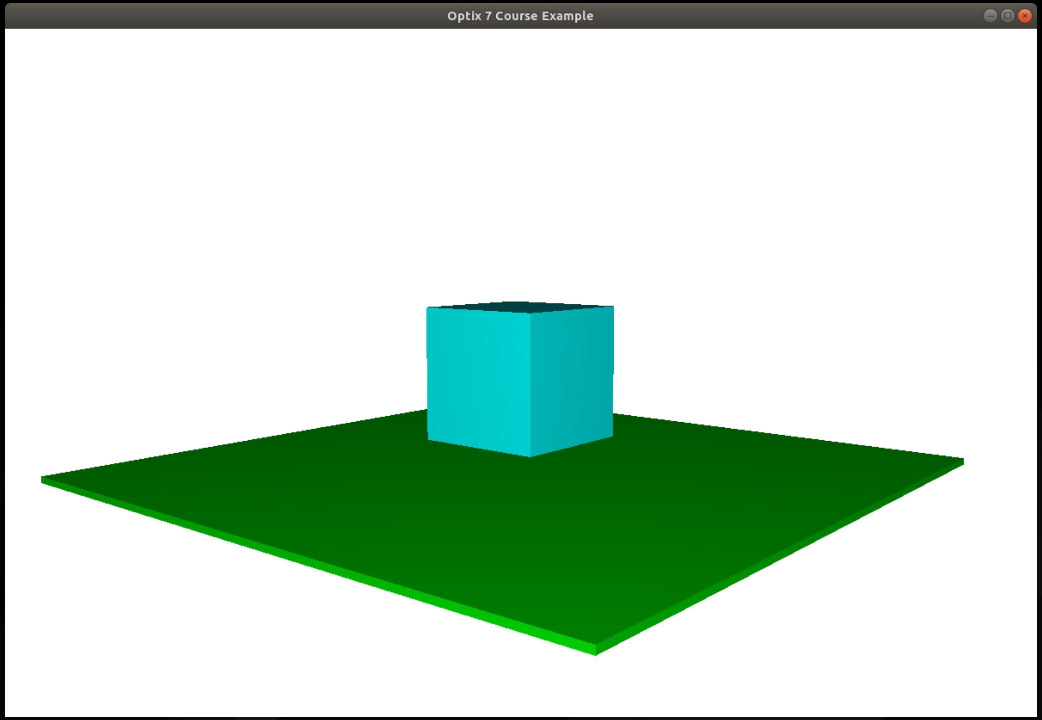
EXAMPLE 6: MULTIPLE INDIVIDUAL MESHES

MULTIPLE TRIANGLEINPUTS (IN ACCEL) AND HITGROUP RECORDS (IN SBT)

Before: one mesh, one material



Now: two separate meshes, w/ material per mesh



EXAMPLE 7: FIRST REAL MODEL

IF WE CAN DO TWO, WE CAN ALSO DO MORE …

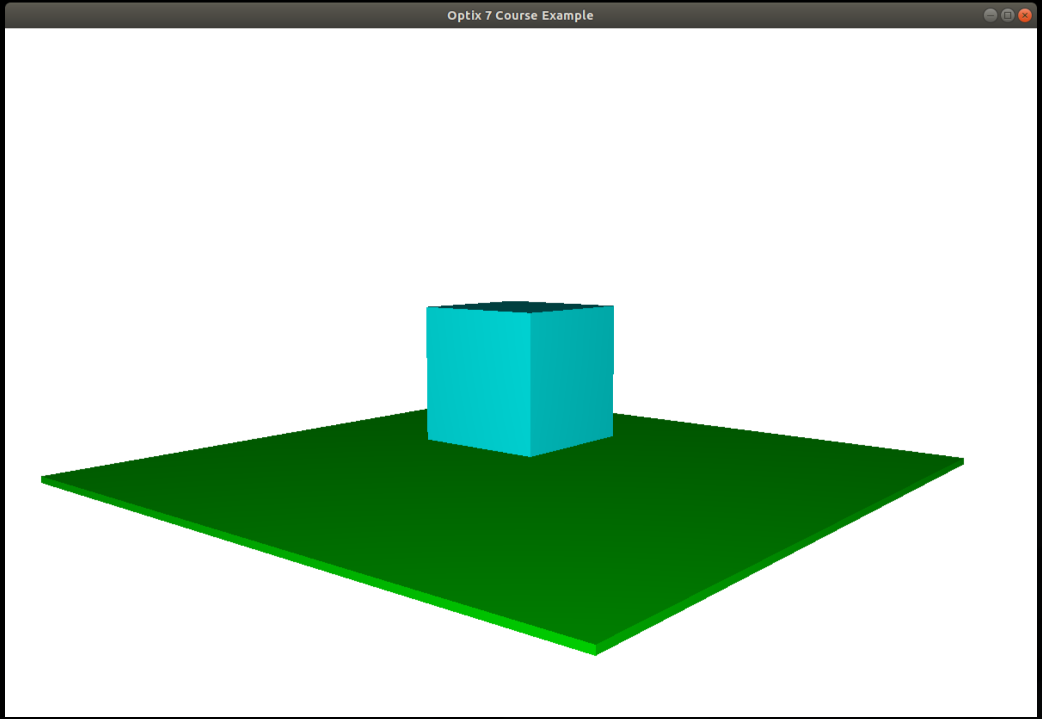
Ok, now it gets really simple:

- Hook up Syoyo’s OBJ parser (https://github.com/syoyo/tinyobjloader)
- Change SampleRenderer from “2” meshes to “N” meshes
- Download “Crytek sponza” from https://casual-effects.com/data/
  - unzip to optix7course root directory – not build dir
- Hard-code a useful camera position into viewer
- Aaaaand …

EXAMPLE 7: FIRST REAL MODEL

LOAD IN SOME REAL MODEL DATA (CRYTEK SPONZA)

Before: two hardcoded meshes for simple testing



Now: w/ actual sponza model's meshes

EXAMPLE 8: ADDING TEXTURES

ALMOST ALL MODEL DATA IN SPONZA IS THROUGH TEXTURES …

- In Optix pre-7: Dedicated "TextureSampler" class that does all the magic.
- In Optix 7: Low-level control → CUDA TextureObjects
  - Host: cudaTextureObjectCreate()
  - Device: tex2D<…>()

EXAMPLE 8: ADDING TEXTURES

OH-KAY: LET'S ADD SOME TEXTURES

1) Create cudaTextureObject for each texture in the model

```
// example8/SampleRenderer.cpp
std::vector<cudaArray_t> textureArrays(numTextures);
std::vector<cudaTextureObject_t> textureObjects(numTextures);
for (each texture) {
   // create 2D array of pixels
   cudaChannelFormatDesc channels = { … };
   cudaMallocArray(…);
   // upload pixels
   cudaMemcpy2DtoArray(…);
   // create texture object
   cudaResourceFormatDesc desc = { … };
   cudaCreateTextureObject(&textureObjects[texID],…);
}
```

EXAMPLE 8: ADDING TEXTURES

OH-KAY: LET'S ADD SOME TEXTURES

2) Add cudaTextureObject to hitgroup record

```
// our Raygen SBT record (alignment is required)
struct __align__(OPTIX_SBT_RECORD_ALIGNMENT) HitgroupRecord
{
   // all the stuff we had before:
   char header[OPTIX_SBT_HEADER_SIZE];
   …
   // plus our new texture object
   cudaTextureObject_t texture;
   bool                hasTexture;
};
```

cudaTextureObject_t works on both host and device!

EXAMPLE 8: ADDING TEXTURES

OH-KAY: LET'S ADD SOME TEXTURES

3) Fill that in during SBT creation

```
// when filling in per-mesh hitgroup record:
…
Material &material = …;
if (material.textureID == -1)
   hgRecord.hasTexture = false;
else {
   hgRecord.hasTexture = true;
   hgRecord.texture = textureObjects[material.textureID]
}
…
```

EXAMPLE 8: ADDING TEXTURES

OH-KAY: LET'S ADD SOME TEXTURES

4) Use CUDA's tex2D() to access that in closesthit program

```
// in __closesthit__radiance() program:
…
vec3f diffuseColor = mesh.color;
if (mesh.hasTexture)
  diffuseColor *= (const vec3f&)tex2D<float4>(mesh.texture);
…
```

CUDA does all the magic…

EXAMPLE 8: ADDING TEXTURES

OH-KAY: LET'S ADD SOME TEXTURES

5) App: Hook up a texture reader

- E.g., use sbt_image.h from https://github.com/nothings/stb
- … then hook this up to model reader (Model.cpp)
- (for this example, this was the Lion's share of the work …)

Aaaand … that's it!

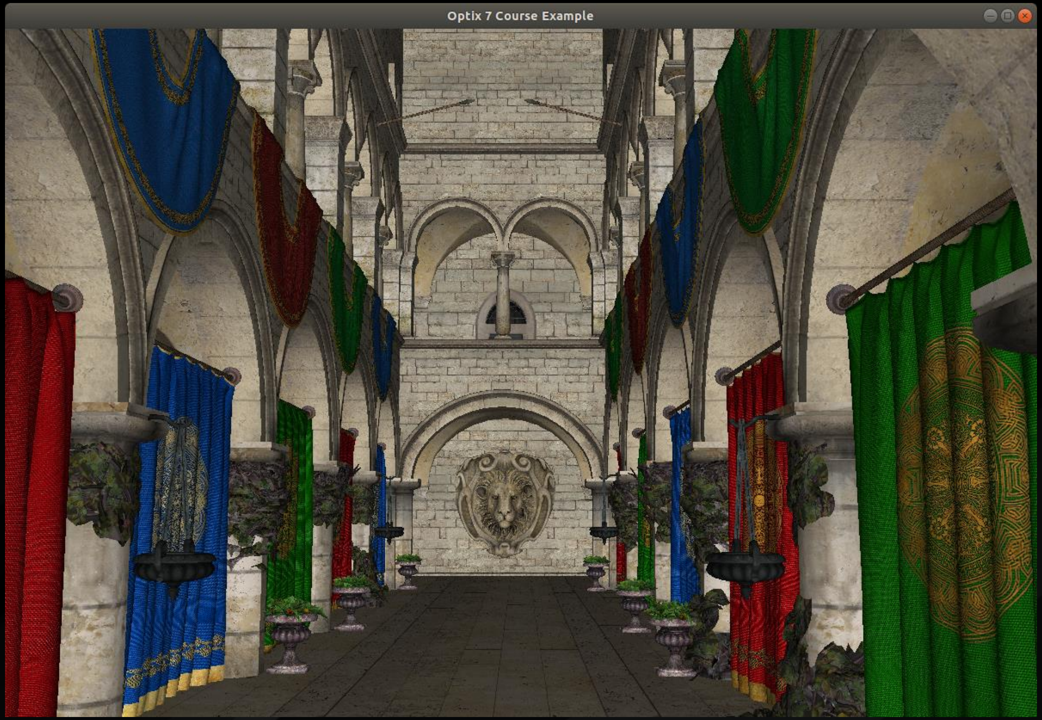
- No changes to STB, accel, pipeline, etc

EXAMPLE 8: ADDING TEXTURES

USE CUDA TEXTURE API (TEX2D(), TEXTURE OBJECTS) FOR TEXTURING

Before: “random” material data

Now: w/ material data and textures



EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING … AT LEAST SOME SHADOWS?

EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING ... AT LEAST SOME SHADOWS?

1) Add a second ray type

```
// in shared header file (visible on both host and device)

enum { RADIANCE_RAY_TYPE=0, SHADOW_RAY_TYPE, RAY_TYPE_COUNT };
```

EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING … AT LEAST SOME SHADOWS?

2) Add closesthit and anyhit programs for this new ray type

- For shadow rays, we use PRD to track how much the shadow ray gets occluded (vec3f)
- all the work happens in *anyhit* (closesthit and miss actually do nothing for shadow rays)
- For this simple example, assume all surfaces are fully opaque:

```
// anyhit for shadow rays:
extern "C" __global__ void __anyhit__shadow()
{
   // set shadow ray to "fully occluded"
   *getPRD<vec3f>() = vec3f(0.f);
   // and kill the ray – no need to do any more traversal on it
   optixTerminateRay();
}
```

EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING … AT LEAST SOME SHADOWS?

2) Add closesthit and anyhit programs for this new ray type

- For shadow rays, we use PRD to track how much the shadow ray gets occluded (vec3f)
- all the work happens in *anyhit* (closesthit and miss actually do nothing for shadow rays)
- For this simple example, assume all surfaces are fully opaque:

```
// anyhit for shadow rays:
extern "C" __global__ void __anyhit__shadow()
{
    // set shadow ray to "fully occluded"
    *getPRD<vec3f>() = vec3f(0.f);
    // and kill the ray – no need to do any more traversal on it
    optixTerminateRay();
}
```

Mind this is for shadow rays types

EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING … AT LEAST SOME SHADOWS?

2) Add closesthit and anyhit programs for this new ray type

- For shadow rays, we use PRD to track how much the shadow ray gets occluded (vec3f)
- all the work happens in *anyhit* (closesthit and miss actually do nothing for shadow rays)
- For this simple example, assume all surfaces are fully opaque:

```
// anyhit for shadow rays:
extern “C” __global__ void __anyhit__shadow()
{
    // set shadow ray to “fully occluded”
    *getPRD<vec3f>() = vec3f(0.f);
    // and kill the ray … no [illegible] rsal on it
    optixTerminateRay();
}
```

For fully opaque: kill the ray upon first occlusion

EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING … AT LEAST SOME SHADOWS?

3) Hitgroup Program Group now has *two* entries

- One for radiance rays, one for shadow rays

EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING … AT LEAST SOME SHADOWS?

3) Hitgroup Program Group now has *two* entries

4) SBT now has to create *two* records per mesh

- Always first one for radiance rays, using program group for radiance rays
- … and second one for shadow rays, using program group for shadow rays
- In our example, we use same data for both records (you do *not* have to)

EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING … AT LEAST SOME SHADOWS?

3) Hitgroup Program Group now has *two* entries

4) SBT now has to create *two* records per mesh

5) optixLaunch() in raygen changes to

```
// trace ray for TWO ray types
optixTrace(launchParams.traversable,ray,…
           // new: indexing for two ray types:
           /* type of ray we trace: */ RADIANCE_RAY_TYPE,
           /* num ray types:        */ RAY_TYPE_COUNT,
           /* miss program index    */ RADIANCE_RAY_TYPE,
           …);
```

EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING … AT LEAST SOME SHADOWS?

6) __closesthit__radiance() launches a shadow ray (fairly obvious code)

```
// in __closesthit__radiance:
…
// compute shadow ray to a point light
vec3f surfPos = …;
vec3f lightDir = lightPos – surfPos;
…
// new PRD, for shadow ray:
vec3f lightVisibility = 1.f;
packPointer(&lightVisibility…);
// and trace, with shadow ray type
optixTrace(…, surfPos, lightDir, SHADOW_RAY_TYPE, …, SHADOW_RAY_TYPE, …);
// finally: use computed lightvisibility in shading
radianceRayPRD = … + lightVisibility * …;
```

EXAMPLE 9: ADDING (HARD) SHADOWS

SO WHAT ABOUT SOME *REAL* RAY TRACING ... AT LEAST SOME SHADOWS?

6) __closesthit__radiance() launches a shadow ray (fairly obvious code)

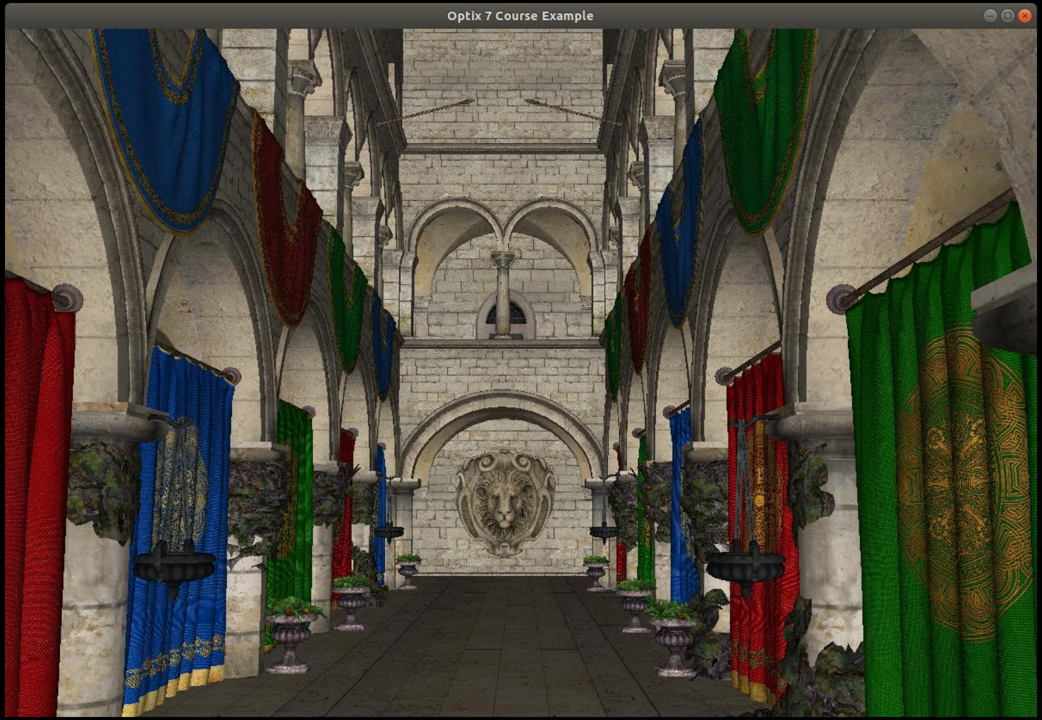
7) hard-code some point light for sponza ...

Aaaand ....

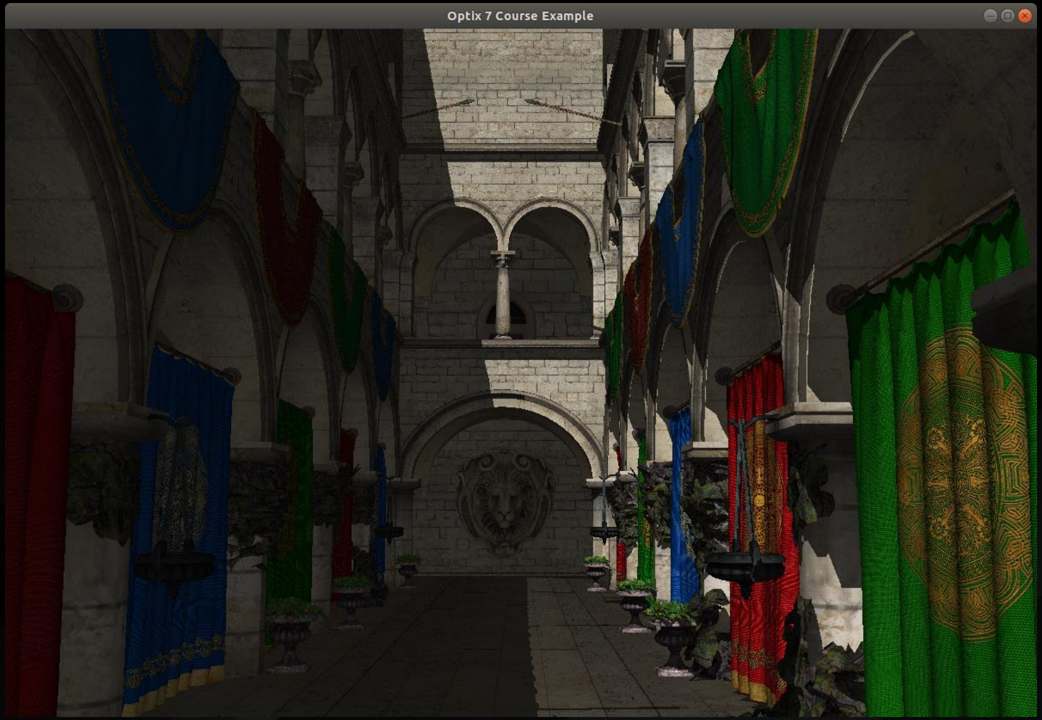
EXAMPLE 9: ADDING (HARD) SHADOWS

ADDING SECOND RAY TYPE FOR SHADOW RAYS

Before: ambient lighting only



Now: w/ shadow from point light source



EXAMPLE 10: SOFT SHADOWS

RANDOM SAMPLING OF AREA LIGHT

From now, it’s all about textbook ray tracing …

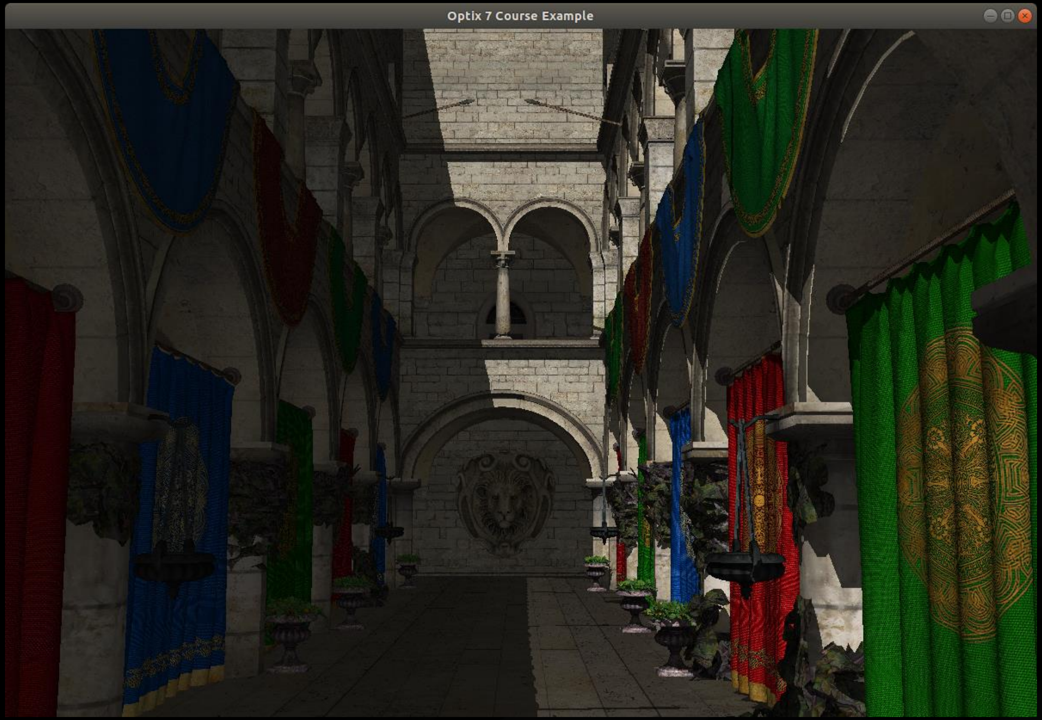
1) Add a quadLight class, and add one in the viewer
2) Add a random number generator (I use a simple one, better ones exist)
3) Add RNG to PRD (so all shaders can share same RNG state)
4) Generate random shadow ray position …
    - And random pixel sample position, too, since we’re at it …

Aaaand ….

EXAMPLE 10: SOFT SHADOWS

RANDOM SAMPLING OF AREA LIGHT

Before: hard shadow from point light



Now: w/ soft shadows from area light

FROM HERE ON: GO PLAY!

THIS WAS ONLY TO GET YOU STARTED – LOTS MORE LEFT TO PLAY WITH

- More fancy shading & path tracing
  - Tip: look at Chris Wyman's DXR tutorial, and do the same here …
  - Also lots of other courses here on sampling, BRDFs, lights, integration, random numbers, …
- Multi-GPU
  - Tip: cudaSetDevice(…) – but careful, different devices might need different SBTs (pointers…)
- Progressive Refinement
  - Tip: add accumulation buffer, accumulate by frame.accumID

FROM HERE ON: GO PLAY!

THIS WAS ONLY TO GET YOU STARTED – LOTS MORE LEFT TO PLAY WITH

- Alpha/cut-out textures
  - Tip: make anyHit_shadow() look up alpha from texture …
- Non-triangular “user geometry”
  - Tip: “IS” (intersection shader) program in hit group (see OptiX SDK for example)
- Instancing
  - Tip: need another accel with “instance” build inputs (again: OptiX SDK)
- Animation
  - Tip: change instance transforms and rebuild “TLAS” (top-level accel struct)

FROM HERE ON: GO PLAY!

THIS WAS ONLY TO GET YOU STARTED – LOTS MORE LEFT TO PLAY WITH

- Denoising
  - Tip: optix actualy ships with a denoiser …
- Async rendering
  - E.g, Launch frame N, launch TLAS update for N+1, denoise N-1, display N-2, …
- Finally: What about some open source “convenience layer” on top of this?
  - Eg, an OS implementation of “GeometryGroups”s, “GeometryInstances” etcpp?

THAT'S IT

GO PLAY WITH IT

- Check it out:
  - https://www.gitlab.com/ingowald/optix7course
    (to go online tomorrow, after last test w/ final Optix 7.0.0 SDK)
- Add to it ...
- Share your results!