

Shading Languages Symposium 2026

Upgrading from GLSL to Slang in the Vulkan Nvpro-Samples



Nia Bickford, NVIDIA

Organised by the **KHRONOS**

Hi! My team has ported dozens of applications to use the Slang shading language.

In this talk:

- Slang in practice
- Porting GLSL
- Ways Slang helped:
 - Pointers
 - Generics
 - Autodifferentiation
 - spirv_asm
 - Reflection
- Improving compile times

2  NVIDIA

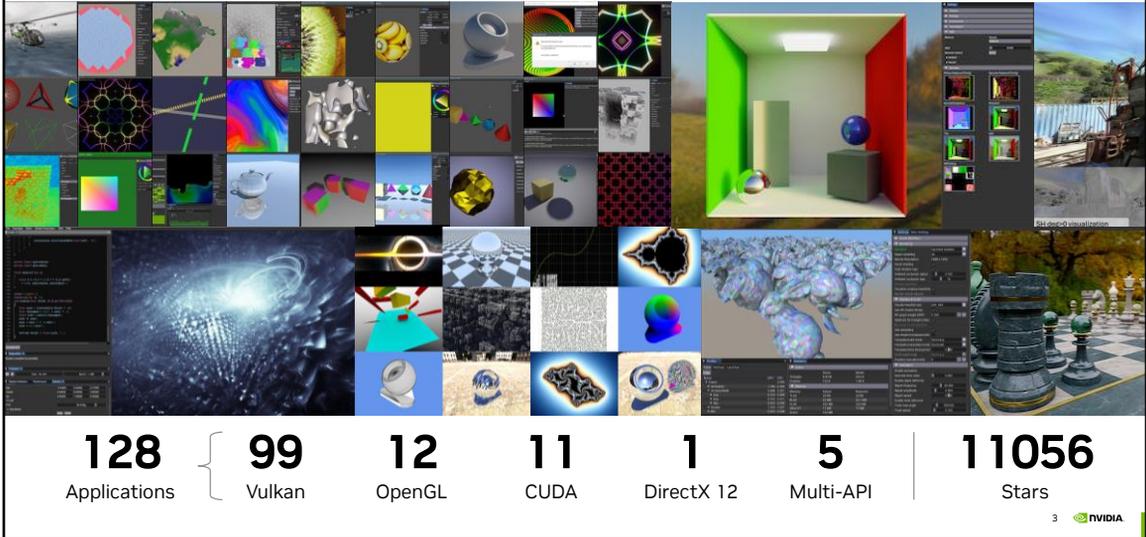
While Shannon talked about Slang from the compiler side, I'm hoping to talk about Slang in practice – what Slang looks like with real codebases.

(click) I'll talk about some neat benefits we get out of it – like pointers, generics, autodifferentiation, SPIR-V assembly, and reflection;

(click) provide tips for efficiently porting and compiling code; and hopefully provide useful experience and ideas to engineers building other shading languages.

The NVIDIA DesignWorks Samples (nvpro-samples)

<https://github.com/nvpro-samples>



At work, my team develops the NVIDIA DesignWorks Samples. Over the past decade, we've built over 100 open-source applications covering all sorts of computer graphics techniques: from ray tracing to Gaussian splatting, texture compression, denoising, video encoding, Mega Geometry, and more. They're written for a variety of APIs; most are in Vulkan, but we've got samples for OpenGL, CUDA, and DirectX as well.

Our samples have been used by developers of all skill levels, all over the world, for all sorts of purposes. Together, our samples have received over 11,000 GitHub Stars.

(Figure from executables produced by build_all's INSTALL target, plus 1 for vk_video_samples, 1 for vk_minimal_latest, and 1 for vk_gaussian_splatting)

📄 Sample Catalog

Graphics & Rendering

Sample	Description	Image	GLSL	Slang
barycentric_wireframe	Single pass solid-wireframe rendering using <code>gl_Barycentric</code>		✓	✓
gtrf_raytrace	gTRF scene loading with path-tracing renderer		✗	✓
image_ktx	KTX image display with tonemapping post-processing		✓	✓
image_viewer	Image loading with zoom and pan functionality		✓	✓
line_stipple	Dashed line rendering with stipple pattern		✓	✓
mesh_shaders	Basic mesh shaders without task shader (baseline)		✓	✓

vk_mini_path_tracer

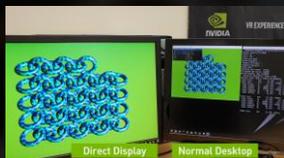
- Beginner path tracing tutorial
- Used during Circle C++ development

vk_mini_samples

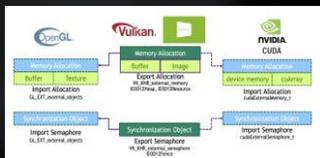
- Wide variety of topics



Many of our readers are computer graphics professionals, and for them we have samples on a wide variety of topics, like the 29 mini-samples written by Martin-Karl Lefrancois.



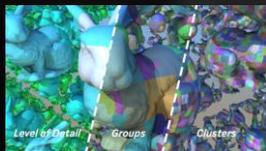
gl_render_vk_ddisplay



gl_cuda_simple_interop



vk_device_generated_cmds



vk_lod_clusters



vk_denoise_nrd



vk_denoise_dlssrr



vk_optix_denoise

vk_mini_path_tracer

- Beginner path tracing tutorial
- Used during Circle C++ development

vk_mini_samples

- Wide variety of topics

“How do I...”

- do GL/CUDA/Vulkan interop?
- use Device-Generated Commands?
- build acceleration structures faster?
- denoise images?

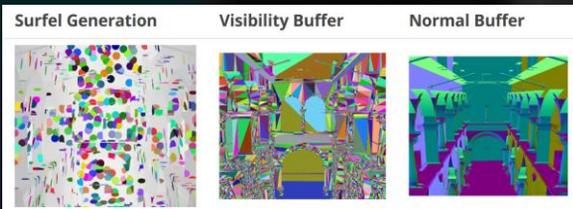
Sometimes a developer will come to us with a problem – like “how do I do interop between GL and Vulkan?” – we’ll write sample code showing how to do that, and then release it so people can learn from it.



A Production-Quality GPU Path Tracer Implementing Four Complementary SIGGRAPH Papers

University [Papers](#) Course [CIS 5550](#) Platform [Vulkan 1.3](#) Shader [Slang](#)

MatForge: <https://github.com/MatForge/MatForge>
Forked from [vk_gltf_renderer](#)



SurfelPlus: <https://github.com/MatForge/MatForge>
Forked from [vk_raytrace](#)

vk_mini_path_tracer

- Beginner path tracing tutorial
- Used during Circle C++ development

vk_mini_samples

- Wide variety of topics

“How do I...”

- *do GL/CUDA/Vulkan interop?*
- *use Device-Generated Commands?*
- *build acceleration structures faster?*
- *denoise images?*

Used for prototyping:

- MatForge
- SurfelPlus

And our samples are easy to modify as well; university students have used them to build projects like [MatForge](#) and [SurfelPlus](#).

Nvpro-Samples' Philosophy

Informs + constrains how we use Slang:

- Clear to read.
- Close to the metal.
- Fast.

⇒ Show you how to **get the best out of your graphics API**.

8 

All our samples have a sort of unifying philosophy, which informs and constrains how we use Slang.

- First, they should be *clear to read*.
- Second, they should be *close to the metal*. In the parts that matter to each sample, you should be seeing raw Vulkan code. No deep abstractions you have to dig through; no engines, no RHIs; what you see is what it does.
 - For things that aren't important to the sample – such as swapchain creation in an SSAO sample – we're OK with abstractions, but they should still be easy to peel away.
- Most importantly, because we're a high-performance graphics team, it needs to be *fast*. For this, Slang's performance – as good as GLSL or better – is really important.

So, in other words, our goal (**click**) is to show you how to *get the best* out of the graphics API you're using.

Why did we choose Slang?

- 2023-2024:
 - Nia experimented with Slang's autodiff for texture compression
 - Martin-Karl ported vk_mini_samples from **GLSL** to **Slang** and **HLSL**
- Compared pros and cons of all 3 languages
 - We want a shading language that works + developers like
 - At the time, Slang was a research project
- Slang had these killer features:

[vk_mini_samples](#) / [samples](#) / [ray_trace](#) / [shaders](#) / 

 [mklefrancois](#) Adding HLSL and Slang shading support

Name
..
device_host.h
rh_bindings.h
payload.h
raytrace.hlsl
raytrace.rchit
raytrace.rgen
raytrace.rmiss
raytrace.slang

9 

We started looking into using Slang in 2023-2024. I had been using Slang's automatic differentiation feature to do some GPU-accelerated texture compression research in 2023. Later that year Martin-Karl really kicked it off by [porting his vk_mini_samples](#) from GLSL to Slang and HLSL, to compare the pros and cons of all 3 languages.

(click) Now, we do work at NVIDIA, but we *do* want to choose a shading language that worked for us, and that developers will like. At the time, Slang was a research project, after all; it hadn't been adopted by Khronos yet. **(click)** But it had a few features that were really appealing for us.

Pointers!

```
GLSL: gltf_pathtrace.comp.gls1 + nvpro_core
struct RenderNode { ... };

struct Scene
{
    uint64_t materialAddress;
    uint64_t renderNodeAddress;
    uint64_t renderPrimitiveAddress;
    uint64_t lightAddress;
    int      numLights;
};

layout(buffer_reference, scalar)
readonly buffer RenderNodeBuf {
    RenderNode _[];
};

// Get RenderNode i
RenderNode renderNode
= RenderNodeBuf(renderNodeAddress)._[i]
```



```
Slang: gltf_pathtrace.slang + nvpro_core2
struct RenderNode { ... };

struct Scene
{
    ShadeMaterial* materials;
    RenderNode*    renderNodes;
    RenderPrimitive* renderPrimitives;
    Light*         lights;
    int            numLights;
};

// Get RenderNode i
RenderNode renderNode = renderNodes[i];
```

10 NVIDIA

The main one was **pointers**. We use GLSL buffer references quite heavily for complex data structures in our glTF and RTX Mega Geometry renderers. But the syntax for pointers is much nicer – and it maps well to what the hardware’s doing, which is a load from an address in VRAM.

Multiple entrypoints!

```
GLSL: vk_gltf_renderer/payload.h
struct HitPayload { ... };
```

```
GLSL: vk_gltf_renderer/pathtrace.rgen.glsl
#include "payload.h"
void main() { ... }
```

```
GLSL: vk_gltf_renderer/pathtrace.rchit.glsl
#include "payload.h"
void main() { ... }
```

```
GLSL: vk_gltf_renderer/pathtrace.rmiss.glsl
#include "payload.h"
void main() { ... }
```

```
GLSL: vk_gltf_renderer/pathtrace.rahit.glsl
#include "payload.h"
void main() { ... }
```

```
Slang: vk_gltf_renderer/raytracer_interface.h.slang
struct HitPayload { ... };
```

```
Slang: vk_gltf_renderer/gltf_pathtrace.slang
#include "raytracer_interface.h.slang"
```

```
[shader("raygeneration")]
void rgenMain() { ... }
```

```
[shader("closesthit")]
void rchitMain() { ... }
```

```
[shader("miss")]
void rmissMain() { ... }
```

```
[shader("anyhit")]
void rahitMain() { ... }
```

Then there's the ability to define **multiple entrypoints in a single file**. In our ray tracing pipeline examples, we often needed to create different GLSL files for ray generation, closest-hit, miss, and sometimes any-hit and intersection shaders. These shaders all need to share the same ray payload, so it's much nicer to define them in a single file instead of splitting them across many files with correspondingly complex includes.

Multiple entrypoints!

GLSL			Slang		
Name	Last commit message	Last commit date	Name	Last commit message	Last commit date
..			..		
denoise.comp.glsl	Bulk Change (2024-07-23)	2 years ago	common.h.slang	Migrate to modern core2 framework	8 months ago
denoise_host.h	Bulk Change (2024-12-03)	2 years ago	dis_sdl.h	Migrate to modern core2 framework	8 months ago
dh_bindings.h	Bulk Change (2024-07-23)	2 years ago	get_bit.h.slang	Migrate to modern core2 framework	8 months ago
get_bit.h	Bulk Change (2024-12-17). Fixes for assets from gltf-Assets (glb, dr...	2 years ago	gltf_pathtracer.slang	Migrate to modern core2 framework	8 months ago
hit_stats.h	Support multiple UV	2 years ago	gltf_raster.slang	Migrate to modern core2 framework	8 months ago
pathtracer.comp.glsl	Bulk Change (2025-01-07). documentation	last year	raytracer_interface.h.slang	Migrate to modern core2 framework	8 months ago
pathtracer.rchit.glsl	Fix largest issue	2 years ago	shader.h	Migrate to modern core2 framework	8 months ago
pathtracer.rchit.sdl	Bulk Change (2024-07-04)	2 years ago	silhouette.comp.slang	Migrate to modern core2 framework	8 months ago
pathtracer.render.glsl	Bulk Change (2025-01-07). documentation	last year			
pathtracer.rmiss.glsl	Bulk Change (2024-07-04)	2 years ago			
rayload.h	Bulk Change (2024-12-03)	2 years ago			
raster.frp.glsl	Adding blurred and solid color background	2 years ago			
raster.vert.glsl	Bulk Change (2024-06-28)	2 years ago			
raster_overlay.frp.glsl	Bulk Change (2024-06-28)	2 years ago			
rt_common.h	Bulk Change (2025-01-16). Adding support for RTR materials, diffuse...	last year			
rt_indirect.h	Bulk Change (2024-12-03)	2 years ago			
rt_layout.h	Bulk Change (2024-12-03)	2 years ago			
rt_nvh	Bulk Change (2024-12-03)	2 years ago			
shadow.rchit.glsl	Bulk Change (2024-12-03)	2 years ago			
shadow.rchit.sdl	Bulk Change (2024-12-03)	2 years ago			
shadow.rmiss.glsl	Bulk Change (2024-12-03)	2 years ago			
silhouette.comp.glsl	Bulk Change (2024-07-30)	2 years ago			

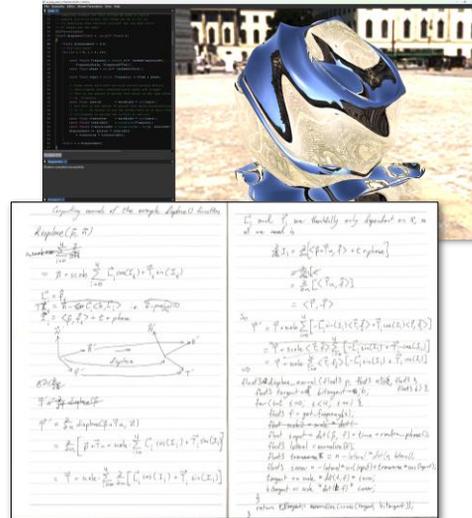
This significantly reduces the visual complexity of our shader folders. Here are the shaders for `vk_gltf_renderer`, a fully-featured glTF path tracer, before and after it was ported to Slang.

Autodifferentiation!

```
float3 displace(float3 p, float3 n) {
    return <complex noise function of p and n>
}

[shader("vertex")]
VsOutput vertexMain(Vertex v) {
    float3 pos = displace(v.pos, v.normal);
    ...
}

[shader("fragment")]
float4 fragmentMain(VsOutput v) {
    // Compute normal manually
    // Don't change `displace()` without updating
    // this code!
    float3 normal
        = <lots of complex manual derivative code>
    ...
}
```



13 NVIDIA

There's also **autodifferentiation**.

My usual example for this is procedural displacement. Suppose I have a shader that displaces vertices, like a Perlin noise function, a wave function, or vertex skinning. Then I also need to compute a new normal vector in the fragment shader for correct shading, which I can do by computing directional derivatives of the displacement function.

(click) I've written code to compute these analytical derivatives by hand many times before, but it always takes quite a bit of algebraic manipulation, it's easy to get wrong, is another code path to test, and is also hard to read. If I need backwards derivatives for optimization, it's another code path.

In addition, if an artist comes to me and asks if we can change the displacement function, then depending on how I've structured the derivative code I might have to re-derive and re-test a lot of it. This slows down iteration times, and weakens the creative process.

Autodifferentiation!

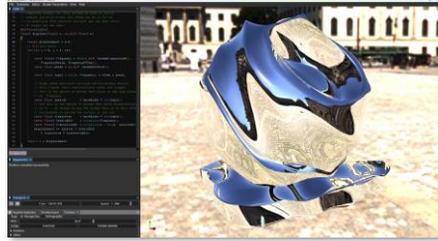
```
[Differentiable]
float3 displace(float3 p, float3 n) {
    return <complex noise function of p and n>
}

[shader("vertex")]
VsOutput vertexMain(Vertex v) {
    float3 pos = displace(v.pos, v.normal);
    ...
}

[shader("fragment")]
float4 fragmentMain(VsOutput v) {
    // Compute displaced tangent and bitangent
    float3 T = fwd_diff(displace)(
        diffPair(v.pos, v.tangent), v.normal).d;

    float3 B = fwd_diff(displace)(
        diffPair(v.pos, v.bitangent), v.normal).d;

    float3 normal = normalize(cross(T, B)); // Done!
    ...
}
```



14 NVIDIA

With Slang, I can mark a function as [Differentiable], and then it'll automatically generate code to compute analytical derivatives for me! Then if I change the displacement function, the normals update automatically.

Fast development!



Finally, Slang has **fast feature development and response times**. Since we're developing samples for new technologies, we often need support for cutting-edge APIs. Often when Martin-Karl or I found a bug or had a missing feature, it'd be fixed or added in a few days, which is really remarkable for a compiler.

So, based on Martin-Karl's research and experience, we decided to port our samples to Slang

Porting GLSL to Slang, Without Much Modification

- Make sure your shader has a `#version` directive
- Mark your entrypoints with `[shader]` attributes
- Domain hull, and mesh shader attributes must be ported to HLSL equivalents, e.g.
 - `layout(vertices = 3) out;` → `[outputcontrolpoints(3)]`
 - `gl_TessLevelOuter` → `SV_TessFactor`
- That's all!
- Docs: <https://docs.shader-slang.org/en/latest/coming-from-gsl.html>



```
#version 400
#extension GL_EXT_shader_image_load_formatted : require

layout (local_size_x = 16, local_size_y = 16, local_size_z = 1) in;

// Uniforms
layout(binding=1) uniform image2D texFrame;
layout(binding=2) uniform image2D texPing;
layout(binding=3) uniform image2D texPong;
layout(binding=4) uniform sampler2D texStrap;
uniform vec2 iResolution;
uniform float iTime;

#define PI 3.1415926535
#define MATTE_Y 0.33

[shader("compute")] // HLSL -> SLANG: Add endpoint attribute
void mainImage()
{
    vec2 thread = gl_GlobalInvocationID.xy;
    if (any(greaterThan(thread, iResolution))) return;
    vec2 fragCoord = vec2(thread);
    fragCoord.y = iResolution.y - fragCoord.y;

    vec2 uv = 2.0 * fragCoord.xy / iResolution.xy - iResolution.x;

    // Matte to widescreen
    // We'll matte again in the final pass.
    // This just saves some processing time.
    vec2 uv2 = 2.0 * fragCoord.xy / iResolution.xy;
    if (abs(uv2.y) > MATTE_Y)
    {
        imageStore(texFrame, int2(thread), vec4(0.0));
    }
}
```

16 NVIDIA

The good news is Slang makes porting code from GLSL to Slang really easy! You only have to do a few small things.

(click) First off, make sure your shader uses the GLSL `#version` directive somewhere. This tells Slang to load in its GLSL compatibility module, which defines type aliases and functions to map GLSL constructs to Slang.

(click) Then, you need to mark your entrypoints with `[shader]` attributes to tell Slang which functions are entrypoints for which shader types.

Most attributes are handled automatically, but if you're porting a **(click)** domain, hull, or mesh shader, which is less common, you have to translate those specific GLSL attributes to HLSL equivalents. It's usually straightforward.

(click) And then you're done!

GLSL

```
Code X
1 #version 460
2 #extension GL_EXT_shader_image_load_formatted : require
3
4 layout (local_size_x = 16, local_size_y = 16, local_size_z = 1) in;
5
6 // Uniforms
7 layout(binding=1) uniform image2D texFrame;
8 layout(binding=4) uniform sampler2D texStarmap;
9 uniform vec2 iResolution;
10 uniform float iTime;
11
12 #define PI 3.1415926535
13 #define MATTE_Y 0.39
14
15 |
16 void main()
17 {
18     uvec2 thread = gl_GlobalInvocationID.xy;
19     if(any(greaterThan(thread, iResolution))) return;
20     vec2 fragCoord = vec2(thread);
21     fragCoord.y = iResolution.y - fragCoord.y;
22
23     vec2 uv = 2.0*(fragCoord.xy-0.5*iResolution.xy) / iResolution.x;
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
26
```

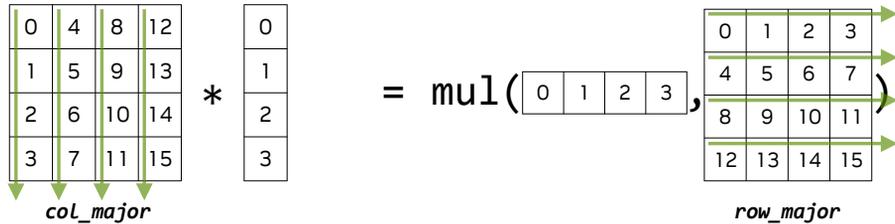
Slang

```
Code X
1 #version 468
2 #extension GL_EXT_shader_image_load_formatted : require
3
4 layout (local_size_x = 16, local_size_y = 16, local_size_z = 1) in;
5
6 // Uniforms
7 layout(binding=1) uniform image2D texFrame;
8 layout(binding=4) uniform sampler2D texStarmap;
9 uniform vec2 iResolution;
10 uniform float iTime;
11
12 #define PI 3.1415926535
13 #define MATTE_Y 0.39
14
15 [shader("compute")]
16 void main()
17 {
18     uvec2 thread = gl_GlobalInvocationID.xy;
19     if(any(greaterThan(thread, iResolution))) return;
20     vec2 fragCoord = vec2(thread);
21     fragCoord.y = iResolution.y - fragCoord.y;
22
23     vec2 uv = 2.0*(fragCoord.xy-0.5*iResolution.xy) / iResolution.x;
24 }
Compile (F3)
Diagnostics X
Shaders compiled successfully.
```

Once we've done that, slangc successfully compiles it.

Matrix Layouts in GLSL vs. HLSL

GLSL	HLSL
$M * v$	<code>mul(v, M)</code>



Slang translates for you: $M * v \rightarrow \text{mul}(v, M)$

19 NVIDIA

One potentially confusing thing you may know is that GLSL and HLSL typically have opposite matrix layouts and conventions for the order in which you multiply vectors and matrices.

In GLSL, matrices are column-major, and matrices usually appear on the left.

But in HLSL, matrices are row-major, and matrices usually appear on the right.

(click) The good news is that Slang's GLSL compatibility layer automatically handles this transposition for you: it'll translate GLSL $M * v$ to HLSL `mul(v, M)`.

Recommendation

- Use row-major layout (-matrix-layout-row-major)
 - **No change needed to your GLSL code**
 - Don't spend time flipping the order or transposing your matrices
 - Using GLM or DirectXMath? memcpy
 - Natural to view in debugger
 - Slightly better access pattern
- Whichever layout you choose, always specify it in the compiler flags
 - Slang's biggest pitfall: slangc defaults to column-major, libslang defaults to row-major
 - Slangc expected to default to row-major in the future
 - SPIR-V annotation will appear opposite what you specify; Slang row-major/SPIR-V Co1Major has the better access pattern.
 - More detail: <https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/a1-01-matrix-layout.html>

20 

However, uploading matrix data can be a bit of a braintwister. If you're looking at SPIR-V it's even more so because Slang also emits that transposed. My team went back and forth on different approaches when we were porting our first samples.

Based on our experience, we have a recommendation: **(click)** specify row-major layout, and then don't mess with your code.

Because Slang's GLSL compatibility layer swaps the order for you, row-major CPU matrices, will show up as row-major HLSL matrices, and as column-major GLSL matrices. This is correct in both APIs.

So, if your GLSL matrix math was working before, it'll work now.

(click) If you're using the GLM or DirectXMath libraries, you can memcpy the matrices to memory. No need for transposition.

(click) And row-major CPU matrices are natural to view in the debugger, **(click)** and have slightly better access patterns for the GPU.

So basically, specify row-major, and then don't touch your code.

(click) The second most important thing is to always specify the matrix layout in the compiler flags. Slang's biggest pitfall at the moment in my opinion is that the library and the executable compilers default to different layouts. The Slang team is hoping to make these both default to row-major in the future, but for now it's best to always specify it.

Porting GLSL to Slang, Without Much Modification

- Make sure your shader has a `#version` directive
- Mark your entrypoints with `[shader]` attributes
- Domain hull, and mesh shader attributes must be ported to HLSL equivalents, e.g.

- `layout(vertices = 3) out;` → `[outputcontrolpoints(3)]`
- `gl_TessLevelOuter` → `SV_TessFactor`

-
- For our samples, we chose the hard route of manually porting code to idiomatic Slang instead!
 - Want to show how to **get the best out of Slang**, not GLSL-over-Slang



```
#version 460
#extension GL_EXT_shader_image_load_formatted : require

layout (local_size_x = 16, local_size_y = 16, local_size_z = 1) in;

// uniforms
layout(binding=1) uniform image2D texFrame;
layout(binding=2) uniform image2D texProg;
layout(binding=3) uniform image2D texMap;
layout(binding=4) uniform sampler2D testSamp;
uniform vec2 iResolution;
uniform float iTime;

#define PI 3.1415926535
#define MATTE_Y 0.33

[shader("compute")] // HLSL -> Slang: Add endpoint attribute
void mainImage()
{
    wvec2 thread = gl_GlobalInvocationID.xy;
    if (any(greaterThan(thread, Resolution))) return;
    vec2 fragCoord = vec2(thread);
    fragCoord.y = iResolution.y - fragCoord.y;

    vec2 uv = 2.0 * fragCoord.xy / iResolution.xy - iResolution.x;

    // Matte to widescreen
    // We'll matte again in the final pass.
    // This just saves some processing time.
    vec2 uv2 = 2.0 * fragCoord.xy / iResolution.xy / iResolution.x;
    if (abs(uv2.y) > MATTE_Y)
    {
        imageStore(texFrame, int2(thread), vec4(0.0));
    }
}
```

And that's it! This is the way I'd recommend people handle porting from large codebases in general. Use `#version 460`, mark your entrypoints, use row-major layouts, and you're done.

(click) For our samples, though, we chose *not* to do this! Instead, we chose to manually port our code over to idiomatic Slang, which looks more like HLSL.

(click) The reason for this goes back to our original principles: we want to avoid abstractions; and since we're teaching how to write samples using Slang, we want to show people how to *get the best out of the Slang* shading language – not how to use GLSL layered on top of Slang.



More Slang Features

- Enumerations
- Generics
- Inline SPIR-V Assembly

22  NVIDIA

In addition to pointers, multiple entrypoints, and autodifferentiation, here are a couple of other neat ways our samples use Slang.

Enumerations

GLSL: nvpro_core/dh_hdr.h

```
#ifndef __cplusplus
#define START_BINDING(a) enum a {
#define END_BINDING() }
#else
#define START_BINDING(a) const uint
#define END_BINDING()
#endif

START_BINDING(EnvDomeBindings)
eHdrBrdf = 0,
eHdrDiffuse = 1,
eHdrSpecular = 2
END_BINDING();
```



Slang: nvpro_core2/hdr_io.h.slang

```
enum EnvDomeBindings {
    eHdrBrdf = 0,
    eHdrDiffuse = 1,
    eHdrSpecular = 2
};
```

- Scoped by default

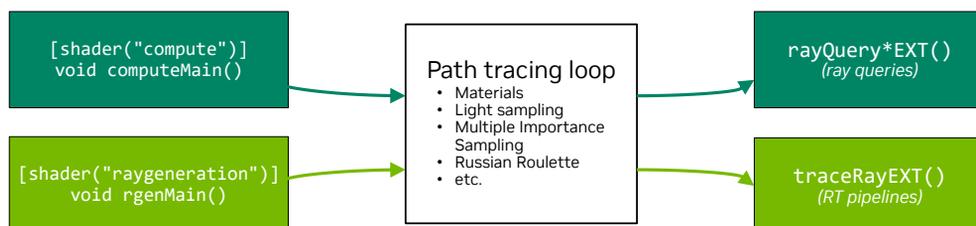
We use enums for things like algorithm types or light types. When we were using GLSL, we had to use some preprocessor magic to make enums work in both C++ and GLSL. Now with Slang, our enums are compatible with C++, and they're scoped by default, which is nice.

Generics in vk_gltf_renderer

- User can switch between **ray queries** and **ray tracing pipelines**



- Affects both which shaders exist, and code deep inside them



24 NVIDIA

We also use Slang's generics in some interesting ways. In some of our ray tracing samples, we want to let the user switch between ray tracing pipelines and ray queries, so they can see which is faster with the sample's shader setup on their GPU.

(click) Typically, ray tracing happens in the innermost parts of the shader: calls to traceRayEXT(), for instance, are surrounded by the rest of the ray-gen or compute shader code to jitter rays, handle multiple bounces, perform multiple importance sampling, and so on.

Generics in vk_gltf_renderer

1. Define an interface for ray tracing backends:

```
interface IRaytracer
{
    void Trace(RayDesc ray, inout HitPayload payload, inout uint seed);
    float3 TraceShadow(RayDesc ray, inout uint seed);
}
```

2. Implement it for ray queries and ray tracing pipelines:

```
struct RayQueryRaytracer : IRaytracer
{
    void Trace(...)
    {
        RayQuery rayQuery;
        rayQuery.TraceRayInline(topLevelIAS, 0, 0xFF, ray);

        while(rayQuery.Proceed())
        {
            ...
        }
    }
}
```

```
struct TraditionalRaytracer : IRaytracer
{
    void Trace(...)
    {
        HitObject hitObj = HitObject::TraceRay(
            topLevelIAS, 0, 0xFF, 0, 0, 0, ray, payload);

        ReorderThread(hitObj);

        ...
    }
}
```

25 

Slang's interfaces and generics give a nice, type-safe way to do this. First, we define an IRaytracer interface. This has a Trace() function to trace a regular ray and store the results in a HitPayload struct, and a TraceShadow() function so the implementation can choose faster ray flags for shadow transmission.

Then we define two implementations of IRaytracer: one which uses ray queries, and one which uses ray tracing pipelines. We can define the closest-hit, miss, and any-hit shaders in the same file as the TraditionalRaytracer struct so our shaders are nicely bundled together. Or, we can use the new HitObject API so we don't need to invoke shaders. And that also lets us use Shader Execution Reordering.

Generics in vk_gltf_renderer

3. Make our path tracing code generic over `IRaytracer`:

```
void processPixel(IRaytracer raytracer, float2 samplePos, float2 imageSize)
{
    ...
}
```

4. Call our path tracing code with `RayQueryRaytracer` or `TraditionalRaytracer`:

```
[numthreads(WORKGROUP_SIZE, WORKGROUP_SIZE, 1)]
[shader("compute")]
void computeMain(uint3 threadIdx: SV_DispatchThreadID)
{
    float2 samplePos = (float2)threadIdx.xy;
    uint2 imageSize;
    outImages[int(OutputImage::eResultImage)]
        .GetDimensions(imageSize.x, imageSize.y);

    RayQueryRaytracer raytracer;
    processPixel(raytracer, samplePos, imageSize);
}
```

```
[shader("raygeneration")]
void rgenMain()
{
    float2 samplePos = (float2)DispatchRaysIndex().xy;
    float2 imageSize = (float2)DispatchRaysDimensions().xy;

    TraditionalRaytracer raytracer;
    processPixel(raytracer, samplePos, imageSize);
}
```

26 

Then, all we have to do is make our path tracing code generic over `IRaytracer`.

Now in our compute shader that uses ray queries, we call our path tracer with a `RayQueryRaytracer` object. And in our ray generation shader that uses ray tracing pipelines, we call our path tracer function with a `TraditionalRaytracer` object. And now we're done. Our code's now clean, type-safe, and doesn't depend on the preprocessor.

Inline SPIR-V Assembly

- Allows you to use SPIR-V features that aren't in the Slang language yet!

```
uint4 MyWaveMatch<T>(T value)
{
    return spirv_asm
    {
        OpCapability GroupNonUniformPartitionedNV;
        OpExtension "SPV_NV_shader_subgroup_partitioned";
        OpGroupNonUniformPartitionNV $$uint4 result $value
    };
}

RWTexture2D<uint> ids;
[shader("compute")]
[numthreads(16, 16, 1)]
void main(uint2 thread: SV_DispatchThreadID)
{
    uint id = ids[thread];
    uint4 ballot = MyWaveMatch(id);
    ...
}
```

SPIR-V Sections

Mode Setting

Debug Information

Annotations

Types, Variables, and Constants

Function Blocks

27 NVIDIA

Sometimes, we develop samples for features that are so new they don't have language support yet. Or, perhaps we're developing a new hardware feature and prototyping new SPIR-V opcodes. Amazingly, the Slang compiler allows you to use new SPIR-V features, even if the Slang language doesn't know about them yet, using *inline SPIR-V assembly*.

This is easiest to describe with an example. **(click)** Imagine if the NonUniformPartition opcode had just been introduced; we wanted to use it, but Slang didn't have it yet. **(click for section placement)**

Full shader example:

```
RWTexture2D<uint> texIn;
RWTexture2D<uint> texOut;
```

```

uint4 MyWaveMatch<T>(T value)
{
    return spirv_asm
    {
        OpCapability GroupNonUniformPartitionedNV;
        OpExtension "SPV_NV_shader_subgroup_partitioned";
        OpGroupNonUniformPartitionNV $$uint4 result $value
    };
}

```

```

[shader("compute")]
[numthreads(16, 16, 1)]
void main(uint2 thread: SV_DispatchThreadID)
{
    uint tid = thread.x + thread.y;
    uint4 ballot = MyWaveMatch(tid);

    uint value = texIn[thread];
    value = WaveMultiPrefixSum(value, ballot);
    texOut[thread] = value;
}

```

Slang's standard library uses spirv_asm

```
slang / source / slang / hsl.meta.slang
Code Blame 32538 lines (30487 loc) · 1.08 MB · ⓘ
7204 // Barrier for writes to all memory spaces.
7205 // @category barrier Memory and control barriers
7206 __glsl_extension(GL_KHR_memory_scope_semantics)
7207 [require(cuda_glsl_hlsl_metal_spirv_wgsl, memorybarrier)]
7208 void AllMemoryBarrier()
7209 {
7210     __target_switch
7211     {
7212         case hlsl: __intrinsic_asm "AllMemoryBarrier";
7213         case glsl: __intrinsic_asm "memoryBarrier(gl_ScopeDevice, (gl_StorageSemanticsShared|gl_StorageSemanticsImage|gl_StorageSemanticsBuffer), gl_SemanticsAcquireRelease)";
7214         case cuda: __intrinsic_asm "__threadfence()";
7215         case metal: __intrinsic_asm "threadgroup_barrier(mem_flags::mem_device | mem_flags::mem_threadgroup | mem_flags::mem_texture | mem_flags::mem_threadgroup_imageblock)";
7216         case spirv: spirv_asm
7217         {
7218             OpMemoryBarrier Device AcquireRelease|UniformMemory|WorkgroupMemory|ImageMemory;
7219         };
7220         case wgsl: __intrinsic_asm "storageBarrier(); textureBarrier(); workgroupBarrier()";
7221     }
7222 }
7223
```

In fact, this is how much of Slang's standard library is defined! If you look at one of the meta.slang files inside Slang's repository, you'll see that most library functions switch depending on the target, then call the intrinsic for the target language or provide SPIR-V assembly.

Two spirv_asm Notes

- spirv_asm can't currently create new opaque Slang types
 - E.g. trying to create EXT hit objects in old Slang versions

- If defining new SPIR-V ops, you must rebuild Slang with new SPIR-V headers
 - So that Slang knows in which region each opcode must be placed

29 

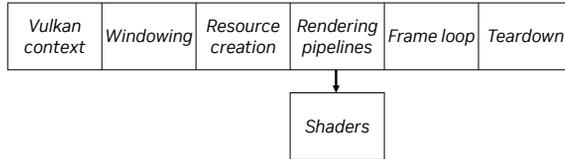
There are two things to note with SPIR-V assembly. First, you can't create new object types this way; Slang doesn't know how to handle opaque SPIR-V types it has no definition for. Secondly, if you're creating new SPIR-V opcodes, you need to define them in the SPIR-V Headers and then recompile Slang, so that Slang knows in which section to put each new SPIR-V opcode.



One of Slang's big features is its powerful shader reflection API, which allows us to write new kinds of apps.

Shader-Defined Renderers

- When writing a Vulkan app, most time is spent on the C++ infrastructure.
 - Pipelines, resources, lifetime management, etc.
 - Shaders are small things that C++ code declares.



- But shaders are (often) the interesting part!
- `vk_slang_editor` inverts this entire structure.
 - The shader defines the renderer!

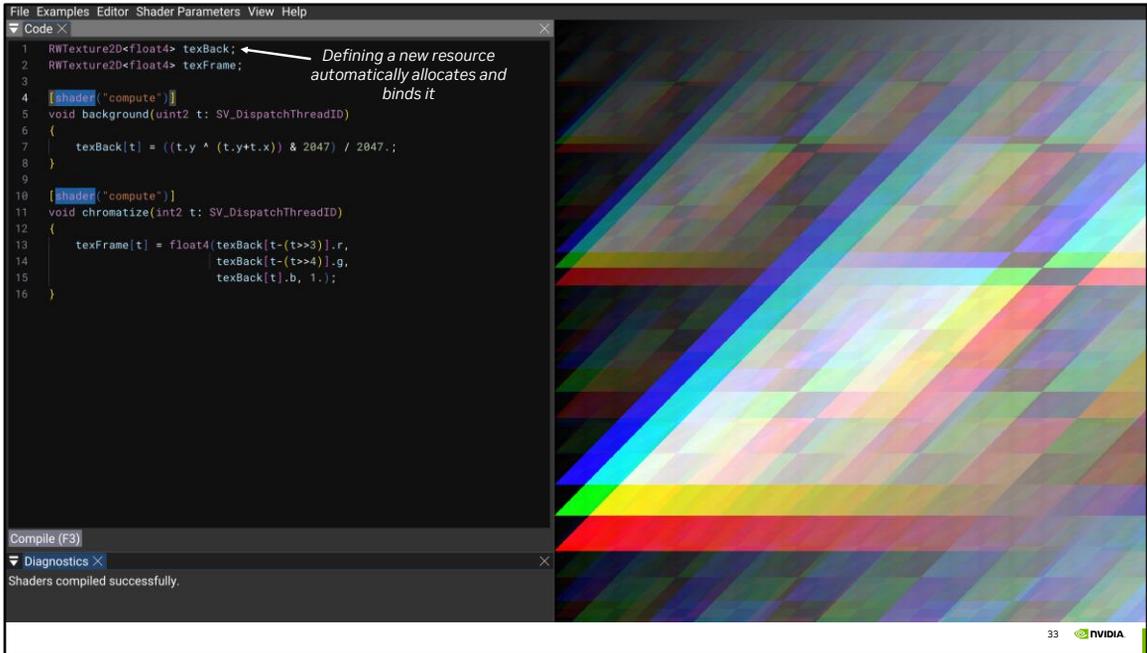
Most of our samples are set up in a very classical way. Typically, when you're writing Vulkan, you spend a lot of time writing C++ code that defines pipelines, resources, and how to manage them. Shaders are the small inner loops within this larger system, with some small bridge headers to match structs between C++ and GLSL.

(click) But shaders are usually the interesting part!

So for SIGGRAPH 2025, we wrote **(click)** `vk_slang_editor`, which is a sample that inverts this entire structure. Instead of C++, you spend your time writing shaders. And based on the Slang shader you write in this interface, the rest of the Vulkan renderer reconfigures itself to run the shader you wrote.

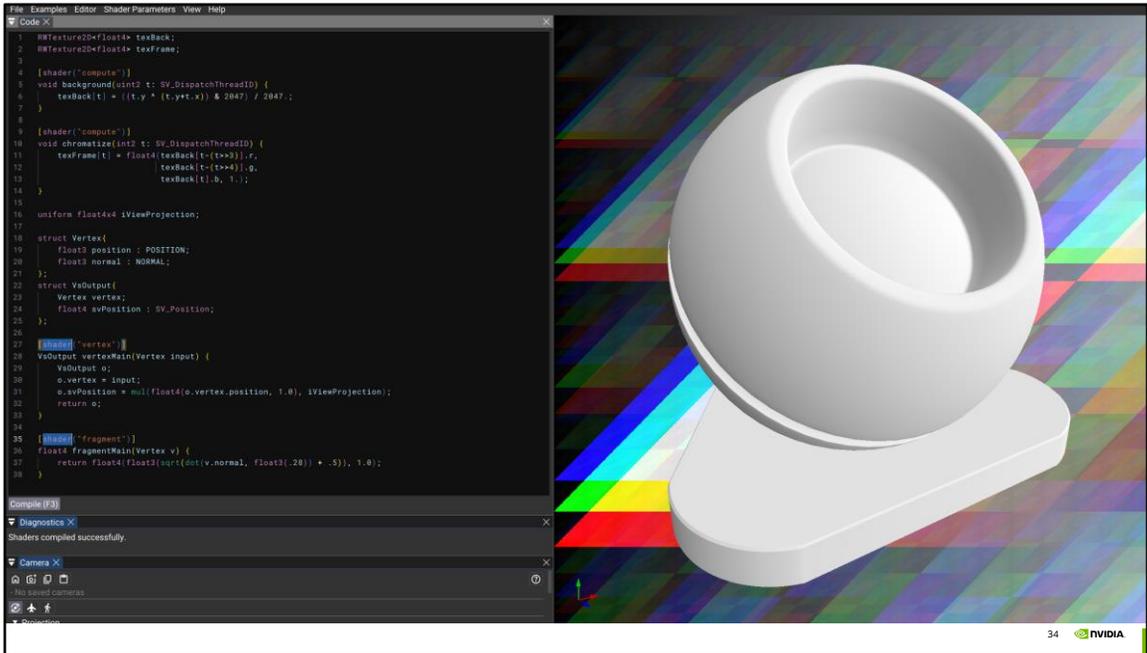


Here's a few examples. Suppose you write a compute shader. Then `vk_slang_editor` will run it as a full-screen compute pass.

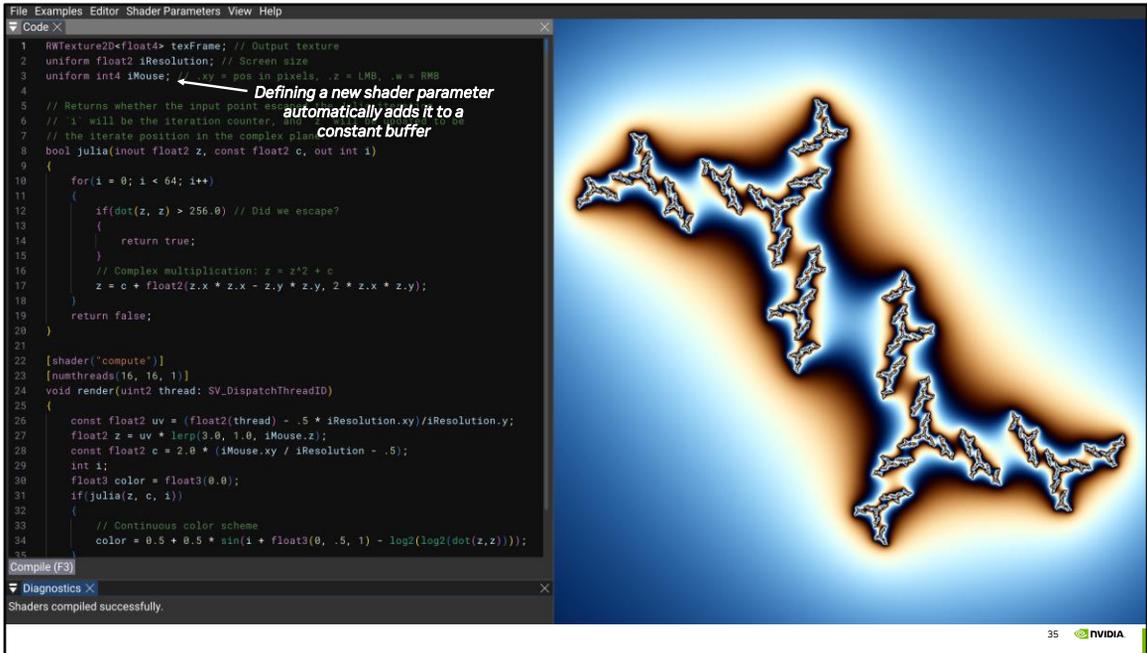


If you want to create a two-pass algorithm, create two compute shaders; we'll run each one in turn.

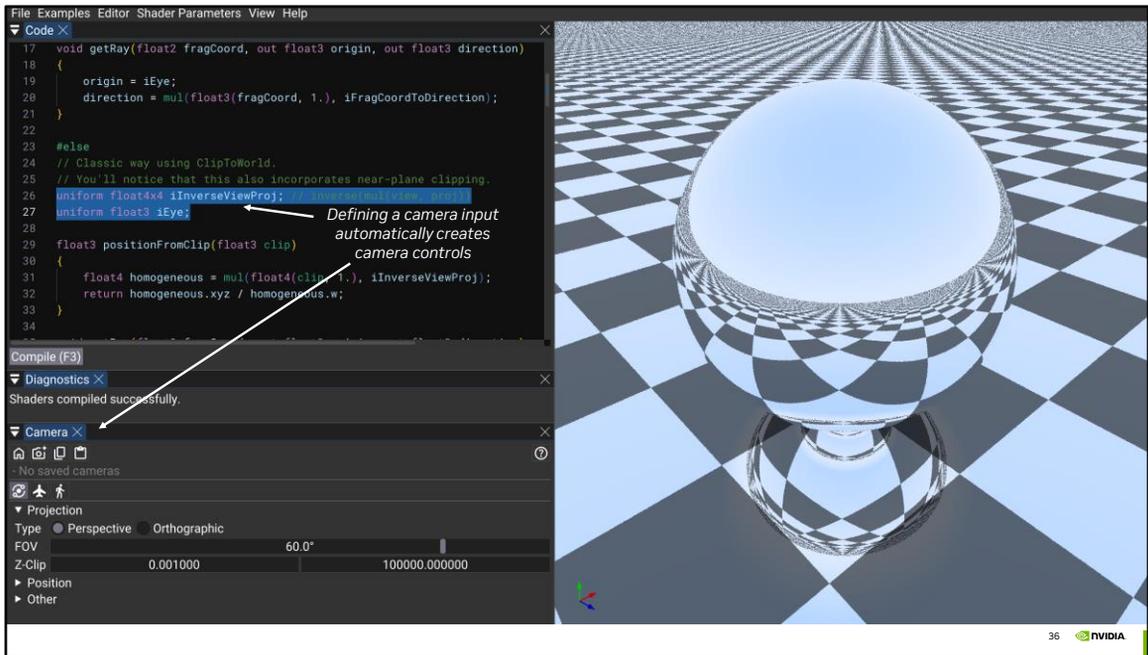
If you need another offscreen backbuffer? **(click)** Define it in the shader, and vk_slang_editor will create it for you.



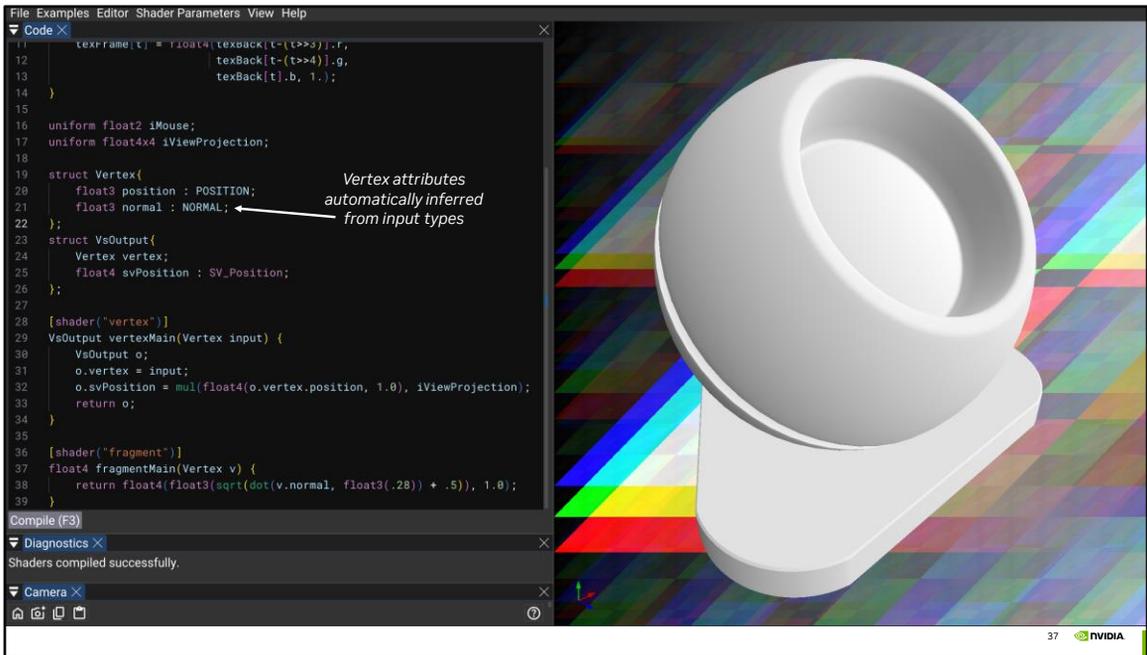
Suppose you add a vertex and a fragment shader. Then `vk_slang_editor` will say “ah, that’s a graphics pipeline”, and combine it into one.



If you need a new shader uniform, like the mouse position, define it with an appropriate name, and we'll start populating it with values. Here we declare `uniform int4 iMouse` at the top of the shader, and now we can use the mouse to control Julia sets.



If you define a camera uniform, we'll even create a camera widget so you can navigate around your shader.



And this even extends to structs. Vertex input attributes, for instance, are defined by your vertex type. If you don't need tangents, we won't bind them. Everything is dynamic.

Why?

- Built for the SIGGRAPH 2025 *Introduction to Slang* hands-on lab
 - Learning a shading language should be about using the language, not building the infrastructure around it

- Inspired by numerous other shader editors:

- Shadertoy
- Gigi
- Bonzomatic
- compute.toys
- FX Composer
- Fragmentarium
- SHADERed



Bonzomatic-Compute's default shader

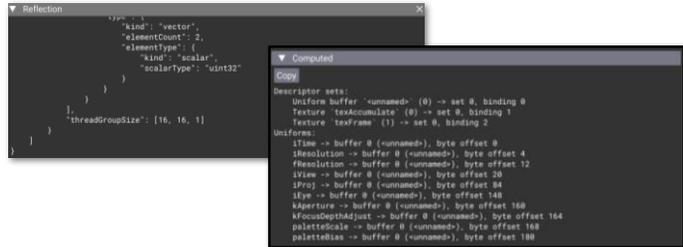
We built this for our SIGGRAPH 2025 course where we taught beginners Slang, because learning a shading language should be about using the language, rather than building the infrastructure around it.

(click) It's inspired by numerous shader editors that came before it, like Shadertoy and the livecoding tool Bonzomatic, which try to minimize the distance between your shader code and the output onscreen.

Slang Reflection

Reflection information includes:

- Resource bindings + info
- Struct layouts + info
- Entrypoints + info
- Much more



- vk_slang_editor walks through this to set up all its pipelines and resources.
- Surprisingly small!

```
47 // edgeBlendLevels);
48 gridlines = max(gridlines, max(gridline.x, gridline.y) * 1/2);
49 }
50 // Evaluates sum(0..N) x^i in [0,3].
51 col = float(0.1) * gridlines;
52 // Defined in C:\shaders\slang\examples\Modules\nr_cubic_solver.slang(26)
53 // public func nr_cubic_solver::evaluateCubic(const float[4] k, float x) -> float
54 {
55     const float cubicY = nr_cubic_solver::evaluateCubic[0], xy.x);
56     // sq's distance estimator from https://iquilezles.com/articles/coherent-ray-tracing/
57     const float dCubicY = nr_cubic_solver::evaluateCubic[1], 3.0 * coefficients[2], xy.x);
58     const float cubicDistance = abs(cubicY - xy.y) / dCubicY
```

- See Theresa Foley's talk, *Getting Started with Slang: Reflections API* online

<https://shader-slang.org/event/2025/06/04/getting-started-with-slang-reflections-api/>

Behind the scenes, this is all possible thanks to Slang's powerful shader reflection features. After a shader has been compiled, we can look at its reflection info to get all of its resources, bindings, structs, struct layouts, entrypoints, functions, and much more. Vk_slang_editor then walks through this info to set up all its pipelines and resources.

(click) Surprisingly, the code to do this is pretty small given that it's so flexible! The code to implement the Vulkan renderer and to do Slang reflection is in fact slightly shorter **(click)** than the code that implements Intellisense in the text editor using the Slang Language Server.

Other Editor Approaches

- <https://shader-slang.org/slang-playground/> is a WebGPU-based editor that uses *custom user attributes*:

```
1 import playground;
2 import rendering;
3
4 [playground::RAND(131072)] ←
5 RWStructuredBuffer<float> randBuffer;
6
```

```
playground.slang
// Initialize a `float` buffer with
// uniform random floats between 0 and 1.
[AttributeUsage(AttributeTargets.Var)]
public struct playground_RANDAttribute
{
    int count;
};
```

+

```
Slang API
struct VariableReflection {
    unsigned int getUserAttributeCount();

    Attribute* getUserAttributeByIndex(
        unsigned int index
    );
}
```

40 NVIDIA

Now, `vk_slang_editor`'s approach isn't the only possible one. Slang Playground is a web-based editor that also uses reflection, but handles this a different way, using *custom user attributes*. For instance, **(click)** here's a custom attribute that says this buffer has a certain size, and should be filled with random values.

(click) The way this works is that every Slang Playground shader imports a module that looks like this. These structs can then be used with attribute syntax. After the shader has been compiled, Slang Playground uses reflection info to get the custom user attribute struct types and values.

Electronic Arts' Gigi editor supports Slang shaders and has an entirely different and much more general approach to defining shaders and pipelines, and does a number of other impressive things. Alan Wolfe will be presenting on Gigi later today, so please make sure to stick around for that.

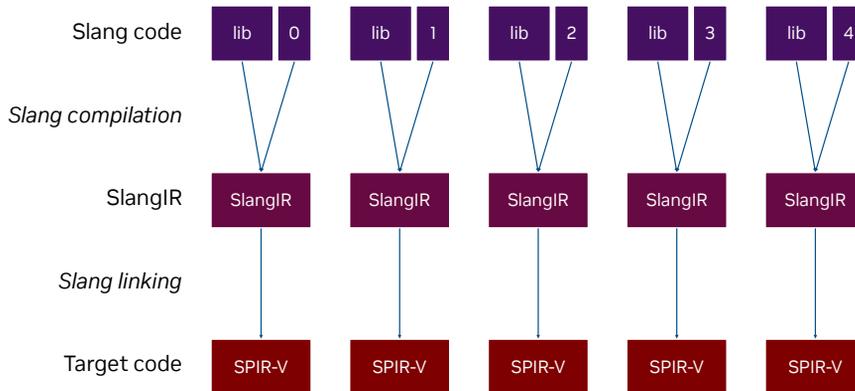
In some sense, the power of Slang's reflection is that it makes this whole range of approaches to shader I/O possible.



Finally, let's talk about modules, and how they can help with compile times.

Modules can Reduce Recompile

- With `#include`, common shader files are compiled many times:

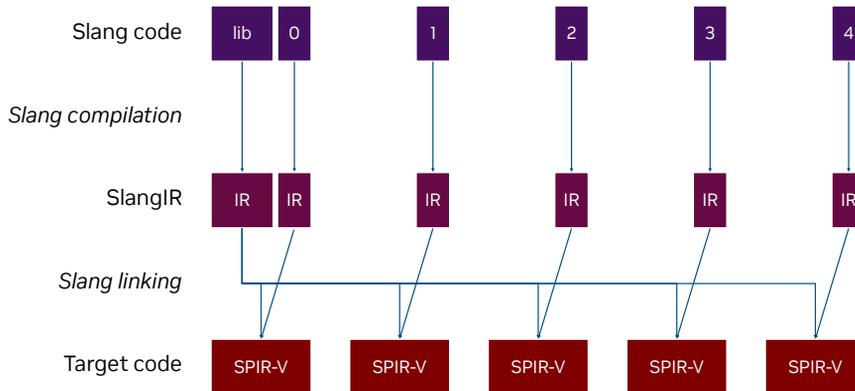


42 NVIDIA

In our core library, `nvpro_core2`, we have common shader files that are used across many samples, like a physically-based BSDF library. Most samples have several shaders, and each one includes different subsets of these files. If a sample has 5 Slang shaders, all of which include our PBR material library, Slang will wind up compiling that PBR material library 5 times, which is slow.

Modules can Reduce Recompile

- With `#include`, common shader files are compiled many times:



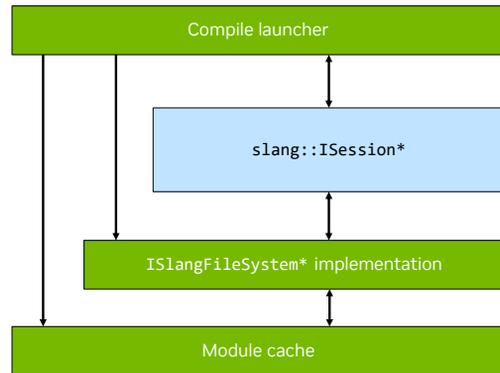
43 NVIDIA

Modules can help with this. Unlike include files, Slang modules can be compiled to SlangIR separately, and then linked together. This can save compile time because now we only need to compile our PBR material library to SlangIR once.

Now, doing this with the Slang library is nontrivial, because we only know what files a Slang shader imported after the shader has been compiled.

A Module Caching System

- Provide ISlangFileSystem
- When Slang searches for a .slang-module:
 - If it's in the cache, provide it
 - Slang will deserialize it into a slang::IModule*
- After Slang finishes compiling a shader:
 - Iterate over slang::ISession::getLoadedModule(i)
 - Serialize and cache each one not already cached
- Also works with multiple threads (assuming the module cache is thread-safe)



44 

One solution is to intercept Slang's IFileSystem interface. Whenever a Slang module is imported, Slang will first look for a precompiled .slang-module with the same name, before looking for the source .slang file. So if we've already compiled a module, we can provide the .slang-module when Slang looks for it. Once we've finished compiling a Slang shader, we can call — to get all the modules it loaded, then take all the ones that aren't in the cache yet and serialize them to the .slang-module binary format.

(click) This also works with multithreaded compiles. While the Slang::IModule* interface can't be shared across different Slang sessions, because it internally references structures inside its session, .slang-module blobs can be, because they're serialized.

Module Caching Benchmark: vk_gaussian_splatting

- Ported vk_gaussian_splatting to use modules
 - Slang v2025.22.1
 - vk_gaussian_splatting: 15 shader files
 - nvpro_core2's core library: 32 header files

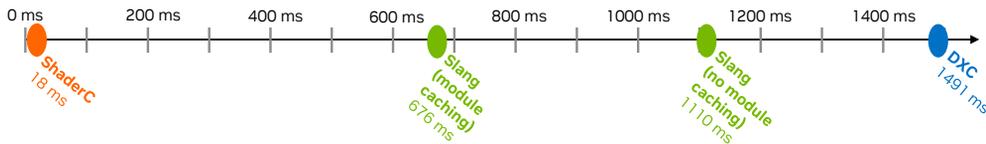
Approach	Slang-to-SPIR-V Time
#include	673 ms
One module per file	597 ms; 11.3% faster
One module for all of nvpro_core2/nvshaders	744 ms; 10.5% slower



We ran a couple of benchmarks, and got decent results from module caching. In one experiment, I ported our Vulkan Gaussian Splatting example to use modules. Turning on module caching gave a 11% compile time improvement. Interestingly, what seems to work best is turning each header file into its own module; if we instead combine all the headers into a single large module, compile times get 10% *slower*, possibly because we're deserializing lots of SlangIR we don't need.

Module Caching Benchmark: Hot-Reloading

- <https://github.com/NBickford-NV/slang-compile-timer>
- 161'720 byte pathtracer from vk_gltf_renderer, 16 modules + 1 main file
- All optimizations and validation turned off, everything in memory, Slang v2026.1.2-5-ga83a63bcb
- Module caching improves hot-reload compile times on this machine and benchmark by 39%



- Also ported this benchmark to GLSL (ShaderC) and HLSL (DirectXShaderCompiler) from Vulkan SDK 1.4.335.0
- All compilers target SPIR-V with fastest compile flags set to make comparisons fair

I also put together an open-source benchmark that simulates *hot-reloading* with and without module caching. For this, I took the main ray query shader from our glTF path tracer along with our core shader library, and ported it to use modules. I figure this is a relatively decent example of a medium-sized codebase; it implements a full PBR material system along with every glTF material extension that existed at the time, and comes out to about 161 KB of shader code.

To simulate hot-reloading, we recompile the shader, acting as if the main shader file had been edited. When module caching is on, we reuse modules for all the other files. On my machine, module caching reduces hot-reload compile times by about 39%.

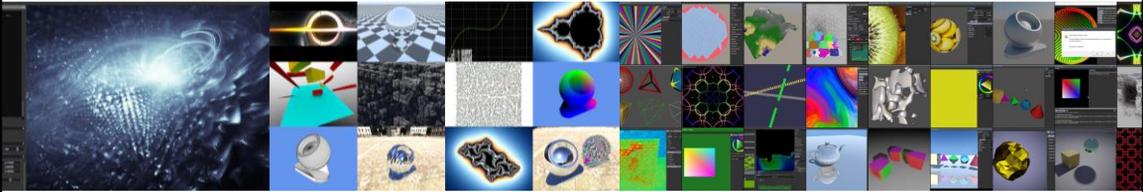
(optional) One reason this might not be as high a reduction as expected is because although module caching lets us avoid compiling Slang to SlangIR, Slang still has to link the SlangIR modules together and compile SlangIR to SPIR-V. The Slang team has talked about experimental future work to reduce this time by linking together modules after they've been compiled to SPIR-V.

(click) Finally, I also ported this hot-reload benchmark by hand to GLSL using glslangValidator, and HLSL using DirectXShaderCompiler, simulating an average developer writing code in each of these languages. To make this comparison fair, I had all 3 compile to SPIR-V, and set appropriate compiler flags to make compiles as fast as possible.

(click) Here are the results on this particular benchmark, versions of these compilers, and my machine. You can see that in this case, Slang is faster than DXC, but both are much slower than ShaderC and glslangValidator when it comes to SPIR-V. So Slang's ok, but since my team comes from GLSL, I would *love* to see both Slang and DXC improve their performance and get closer to the left here.

Conclusion

- Slang's **pointers, generics, autodifferentiation**, and more made our code **cleaner and easier to write** while **maintaining high performance**.
- **Porting from GLSL to Slang** is fast: use `#version 460`, mark your entrypoints, and use row-major layout.
- **Inline SPIR-V assembly** lets you use leading-edge features even if they aren't in the standard library yet.
- Slang's **reflection system** is exceptionally detailed and powerful.
- **Module caching can improve compile times**, in addition to modules' other benefits, like encapsulation.



<https://github.com/nvpro-samples>

Thank you!

47 

This about wraps up our whirlwind tour of some of the neat things we've been getting from using Slang in the nvpro-samples.

Slang's pointers, generics, autodifferentiation, and more made our code cleaner and easier to write, while continuing to have the performance we need.

There's **(click)** an easy route to port from GLSL to Slang, which is what I recommend for most projects starting out: if you have a version directive, the GLSL Compatibility Layer will take care of most of it. You only need to mark your entrypoints, fix up some hull and domain shader declarations, and choose a matrix layout.

We talked about some things unique to Slang, like its **(click)** inline SPIR-V assembly, which allows high levels of control and usage of new features, and its **(click)** exceptionally detailed reflection system, which can be used to make exceptionally dynamic renderers.

Finally, we discussed how **(click)** module caching can improve compile

times, although you'll always want to do performance measurement to ensure it benefits your app.

All of our samples are open-source, and you can find them at github.com/nvpro-samples.

(click) Thank you very much!

