WebGPU

WebGPU doesn't currently work in all browsers. It does currently work in Chrome, but not in firefox. See browser support status here: https://caniuse.com/webgpu

You can debug WebGPU with pix. See the "Pix Capture From a Browser" section.

Learning WebGPU

This is a great introduction to webgpu which shows how to implement Conway's game of life using compute shaders: https://codelabs.developers.google.com/your-first-webgpu-app#0

This slide deck talks about the technical details of webgpu: https://docs.google.com/presentation/d/1v4B4lnHIuhZdzAKGQkMZQtnPrlAtMaY5NmjQVfYgimM/mobilepresent?slide=id.p

WebGPU Limitations

You can see the webgpu capabilities of your browser at https://webgpureport.org/

As of writing this, WebGPU:

- does not support raw byte buffers.
- Has very limited support for different formats in read-write storage textures (aka UAVs).
    - Only r32float, r32sint, and r32uint can be read-write storage.
    - These can be used as read only or write only:
        - rgba8(unorm/snorm/sint/uint)
        - rgba16(float/sint/uint)
        - rg32(float/sint/uint)
        - rgba32(float/sint/uint)
    - Note it's missing things like r8 (unorm/snorm/sint/uint)!
    - Details: https://gpuweb.github.io/gpuweb/#plain-color-formats
    - To deal with this, if you have a read/write buffer to something other than the supported formats, it will split it into a read only and write only resource access pin, where the read only pin reads a copy of the resource. This requires you use the tokens **/*$(RWTextureR:pin)*/** and **/*$(RWTextureW:pin)*/** instead of using the pin name directly in the shader. Technically you only need to use RWTextureR, because RWTextureW accesses the original resource, to write to it.
- Does not allow you to create an sRGB format texture with the GPUTextureUsage.STORAGE_BINDING flag (UAV), which means you can't write to them from compute shaders. Since you can't even create them, that means you also can't override their view format to be non sRGB like we can in DX12 for example. To get around this, Gigi will always make and expect textures made with the non sRGB format, even if the _format variable is an sRGB format. When doing a regular texture read (aka an SRV), the generated code will override the view to specify the sRGB format there. This gives the same

behavior as you expect in other code generation targets: UAV read/write to sRGB formats don't do sRGB conversion. SRV reads to sRGB formats do sRGB conversion.

- Is missing features like raytracing and variable rate shading. We will emulate what we can and makes sense to. We will need a table to show what features are supported on which platforms, likely based on which unit tests are expected to run on each platform.
    - Ray tracing emulation is finished and supports both raw vertex data and BVH data.
        - Raw vertex data ray tracing is brute force and is very slow.
        - BVH allows doing O(log(n)) triangle tests instead of O(N) and is essentially as fast as built in raytracing APIs.
        - See the GigiViewerDX12_Documentation file for information about making BVHs. See the AnyHit and simpleRT raytracing javascript unit test logic for information about using BVHs.
- Mesh shaders are not supported in webgpu.
- Variable rate shading is not supported in webgpu.
- Storage textures require that you specify the texture format in the shader, as well as in the cpu side definition of the shader. This is a problem for shaders that are meant to work with a variety of input types, or an unknown input type (like, an imported resource). To work around this, Gigi puts special tokes into the shader for where the format should go, and then at runtime, the generated javascript will replace those strings with the correct format when compiling the shaders. If the format coming in changes, it will re-create the shader related objects for the new format.
- Storage textures cannot be used in vertex shaders. So you can't write to textures from a vertex shader. You also cannot use atomic operations in vertex shaders.
- Draw calls seem fine to use unpadded / unaligned vertex buffers, but if you want to read/write that VB from a shader, it expects it to be padded / aligned. Since you could reasonably want to have a dynamic VB written to by a compute shader, or read VB fields from a raytracing shader or similar, all vertex buffers are padded / aligned, even though that wastes memory. Example: having a vertex buffer with float3 position and float3 normal could be 6 floats per vertex, but this would make it be 8 floats per vertex, aligning each float3 to float4 boundaries.
- You cannot write to a cube map texture from a shader. The workaround for this is to write to it as a 2d texture array instead. We could probably make this happen automatically during the Gigi code generation process, if we rely on people using the RWTextureR and RWTextureW tokens.
- Rasterizing to a slice of a 3d texture doesn't seem to be working. This can be seen if generating code for and running Texture3DRW_PS.
- Sampler addressing mode "Border" is not supported.
- Atomic operations like InterlockedAdd are not supported for textures, they are only supported for buffers and group shared memory, and only for uint32 or int32.
    - To allow atomic operations for WebGPU, you must check the box "Allow Atomic Ops" in the shader resource declaration, or in the structure field declaration. This is used to declare the buffer type or struct field type as atomic in slang (a requirement for wgsl output). It's possible that slang could be modified so that in a legalization

pass, it detects which things are used in atomic operations, and automatically define them as atomic objects. That would get rid of the need for this checkbox.

- Wave ops, like WaveActiveSum need to be called from uniform control flow (no early returns or calling conditionally), because webgpu/wgsl requires that.
- Group shared memory works in the browser (chrome) but doesn't seem to work in node.js yet (dawn). It gives an error when you require subgroups in node.

Learning Slang

Slang is used to transform the hlsl shaders into wgsl. Slang has a lot of very interesting features, including polymorphism and automatic differentiation. You can learn more about it here:

https://shader-slang.org/slang/user-guide/

Slang Limitations

This will be updated as new things are found, or old things are fixed.

- Slang supports atomics, but it doesn't seem to want to translate hlsl intinsics into using atomics. For instance it complains if you use InterlockedMin, even though slang has an atomic min operation https://shader-slang.org/stdlib-reference/types/atomic-0/min.
    - I did some extra hoop jumping to make this work, but wgsl atomics are very limited. See the slang limitations section for more info.
- Slang has declared shader temporaries as const when they weren't and made wgls shader compilation errors. https://github.com/shader-slang/slang/issues/6965
- Slang crashes when you use GetDimensions on buffers sometimes. https://github.com/shader-slang/slang/issues/6842
- Slang will sometimes incorrectly write directly to textures using an index operator instead of the textureStore function, which is a wgsl compile error. https://github.com/shader-slang/slang/issues/6551
- Slang makes struct member functions const by default, requiring you to put a [mutating] attribute on them. I've asked for a feature to make member functions non const by default. https://github.com/shader-slang/slang/issues/6996

An issue with vertex and index buffer struct field locations

When putting a vertex shader input struct through slang to generate wgsl like the below:

```
struct VSInput
{
    float3 position : POSITION;
    float3 normal : NORMAL;
    float2 uv : TEXCOORD0;
    float3 offset : TEXCOORD1;
```

```
    float scale : TEXCOORD2;
};
```

It assigns arbitrary location values:

```
struct vertexInput_0
{
    @location(0) position_1 : vec3<f32>,
    @location(3) normal_1 : vec3<f32>,
    @location(4) uv_1 : vec2<f32>,
    @location(1) offset_0 : vec3<f32>,
    @location(2) scale_0 : f32,
};
```

This is a problem, because the generated code needs to specify the right locations when calling createRenderPipeline, for the vertex.buffers[].attributes field.

If the semantics are left off of the struct definition, slang will order them in order. You can also use [vk::location(1)] to give a specific field a specific location value in the wgsl.

These are nice options, except that the user making the technique is the person who writes the vertex shader input struct and there is no syntax that you can write to satisfy both hlsl and slang / wgsl.

For now, this requires a manual step after webgpu code generation to fix up the locations in the struct definitions, or the shaders.

One possible solution could be to have Gigi generate the vertex input struct, which would let us define it differently when compiling for webgpu. There are complications to think through though, like if Gigi should also write the vertex shader declaration including this input, or not.

In A Browser

The packages that come out of Gigi have a batch file to start a web server using python, and to open the web page.

You can create a place to host static web pages in gitlab using these steps:

- make a new repo. visibility internal.
    a. Settings -> general. "Visibility, project features, permissions" set Pages access to "Everyone"
- go to the Deploy -> Pages
- follow the 4 steps

    a. image: docker.artifactory.ea.com/alpine:latest
    b. check "The application files are in the 'public' folder
    c. no installation steps
    d. Build steps: echo 'Nothing to do...'
- Going back to Deploy -> Pages will give you the URL of where your stuff lives. Your page will show up there after the deployment job has finished.

Pix Capture From a Browser

I got these steps from https://gist.github.com/Popov72/41f71cbf8d55f2cb8cae93f439eee347.

You need to download Chrome Canary, which is fine to install next to regular chrome: https://www.google.com/chrome/canary/

In Pix, set the path to executable to the chrome canary executable. Mine is at: C:\Users\awolfe\AppData\Local\Google\Chrome SxS\Application\chrome.exe

The working directory is set to that folder, such as: C:\Users\awolfe\AppData\Local\Google\Chrome SxS\Application

Here are the command line arguments I use:
--disable-gpu-sandbox --disable-direct-composition --enable-dawn-features=emit_hlsl_debug_symbols,disable_symbol_renaming

The pix checkboxes must have “Launch for GPU Capture” and “Force D3D11On12” turned on, and “Launch Suspended” and “Enable DRED logging” off.

From there, you press launch and it opens up chrome canary. Navigate to the URL for your technique (such as 127.0.0.1/index.html) and then you can click capture.

After taking a capture, my work was in Graphics Queue 1, not the default Graphics Queue 0. Also, about 1 in 3 times, the capture doesn’t contain my technique. I think it’s because the technique may not run every frame.

The generated code has debug markers in it. See this webpage for information on enabling them to see them in the pix captures: https://dawn.googlesource.com/dawn/+/refs/heads/chromium/4479/docs/debug_markers.md

In Node.js

Node.js is a way to run javascript outside of a browser.

Using webgpu in node.js can be a nice way to run webgpu code headlessly, for automated testing, or GPU powered utilities.

To install node.js: https://learn.microsoft.com/en-us/windows/dev-environment/javascript/nodejs-on-windows

Learning plain node.js

Here is a nice 15 minute video about node.js for absolute beginners: https://www.youtube.com/watch?v=ENrzD9HAZK4

Here is a tutorial for node.js (without webgpu): https://learn.microsoft.com/en-us/windows/dev-environment/javascript/nodejs-beginners-tutorial

TL;DR you can put this into a file named "index.js" and then in a command line, run either "node index.js" or "node .". Using the second command will look for the default named file, which is index.js.

```
var msg = 'Hello World';

console.log(msg);
```

WebGPU in node.js

There is a nice example here that renders a triangle, reads back the texture, and saves it out as a .png at https://github.com/greggman/node-webgpu/blob/main/example/example.js

Save that file into an empty, then open a command line in that folder and run this to initialize a package.json file:

"npm init -y"

Next, run this to download the required node.js libraries for this demo:

"npm install webgpu pngjs"

Next you can run the demo with this:

"node example.js"

An output.png of a red triangle on a grey background should exist afterwords if all ran successfully.

If you wanted to check this into source control and share it with other people, the "node_modules" folder usually is not included in source control. When a person pulls it out of source control without that folder, they can run this command to download the required node.js libraries:

"npm install"

Running WebGPU Unit Tests

The first step is to generate the code for the WebGPU unit tests. You do this by running MakeCode_UnitTests_WebGPU.bat. If there are no code changes, you can consider that a success.

Running the unit tests themselves can be done by running **_GeneratedCode\UnitTests\WebGPU\RunTests.py**. The tests require node.js be installed. If the tests report only successes and no failures, the unit tests have passed.