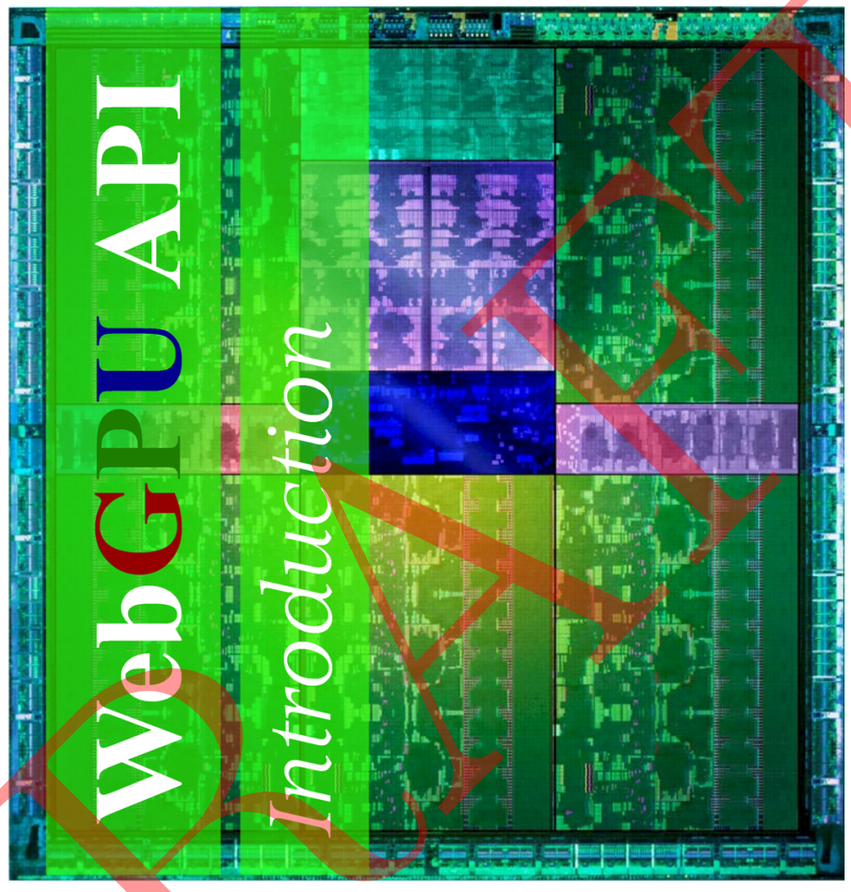


WEBGPU API INTRODUCTION

Kenwright

BOOK TITLE:
WebGPU API: An Introduction
ISBN-13: 979-8-7958-6128-9

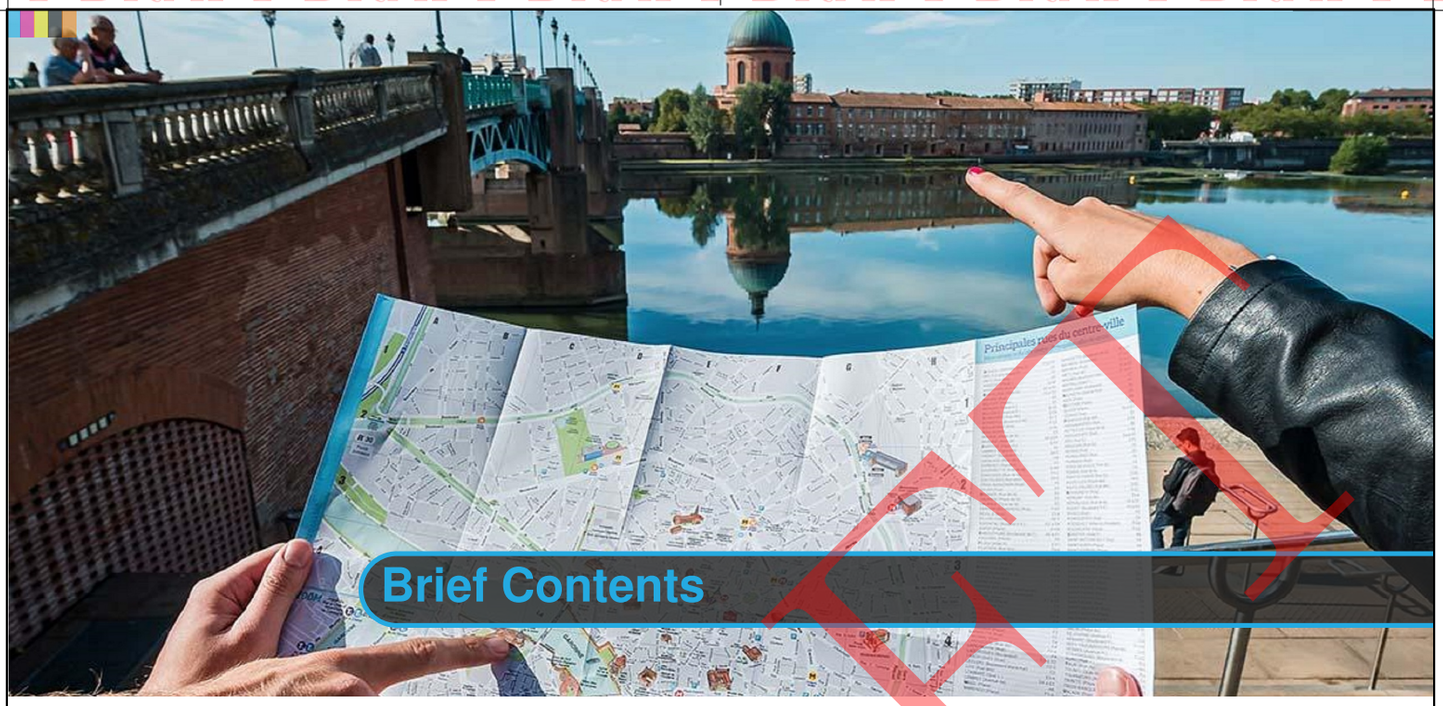
Revision: 06290912022

Preface

Today's web-based programming environments have become more multifaceted for accomplishing tasks that go beyond 'browsing' web-pages. The process of developing efficient web-based programs for such a wide array of applications poses a number of challenges to the programming community. Applications possess a number of workload behaviors, ranging from control intensive (e.g., searching, sorting and parsing) to data intensive (e.g., 3d-graphics, image processing, simulations and data mining). Web-based applications can also be characterized as compute intensive (e.g., iterative methods, numerical methods, and financial modeling), where the overall throughput of the web application is heavily dependent on the computational efficiency of the underlying hardware. Of course, no single architecture is best for running all classes of workloads, and most applications possess a mix of the workload characteristics. For instance, control-intensive applications tend to run faster on the CPU, whereas data-intensive applications tend to run fast on massively parallel architectures (like the GPU), where the same operation is applied to multiple data items concurrently. To extend and support these various workload classes so that browser-based applications wouldn't be hindered, a new generation of API needed to be developed (open the door for developers so that they can access the power of new hardware/technologies). One example of this, is the WebGPU API which exposes the capabilities of GPU hardware for the Web. This text is intended to help you get started with the WebGPU API, while understanding both the HOW and WHY behind it works, so you can create your own solutions. The material in this book is designed to teach you the new WebGPU API for graphics and compute techniques without any prior knowledge. All you need is some JavaScript experience and preferably an understanding of basic trigonometry. Whether you're new to graphics and compute development or an old pro, everyone has to start somewhere. Generally, that means starting with the basics which is the focus of this course. You'll learn through simple, easy-to-learn hands-on exercises that help you master the subject. It does this by using multiple task-based activities and discussions which complement and build upon one another.

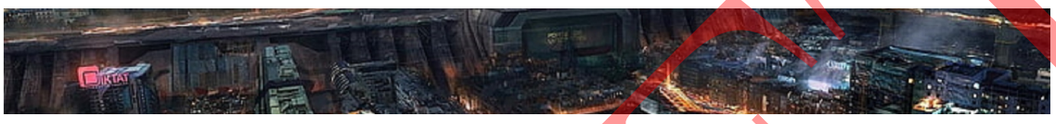
- Understand the core principles of the WebGPU API
- Ground yourself with compute and graphical principles
- Familiarize yourself with the WebGPU API methods
- Create graphical and compute applications using JavaScript and the WebGPU API
- Use and understand application/shader communication/data transfers
- Learn about the WGSL shader language

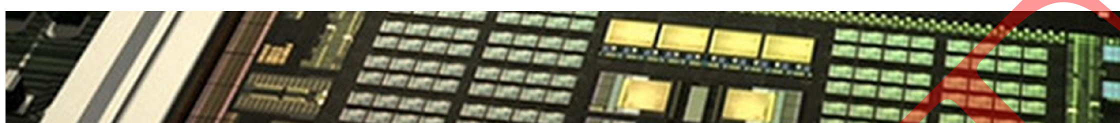
You'll learn how to leverage the WebGPU API to build interesting and useful web-based applications. This text will give you a number of examples that you can test and run to help you see the real power of the WebGPU API. Hopefully, after reading this text, you'll embrace the WebGPU API and continue to explore and take advantage of benefits it offers.

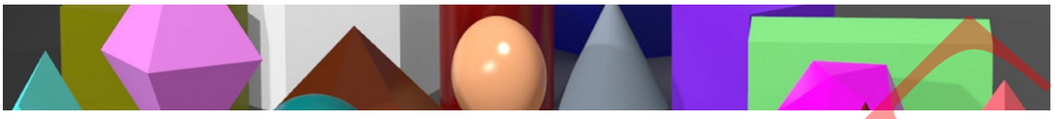


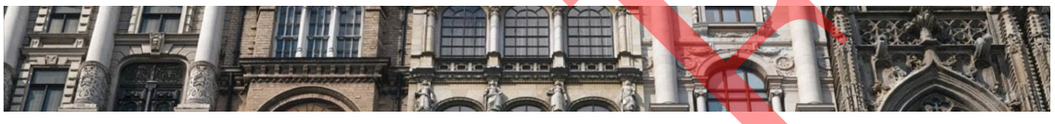
Brief Contents

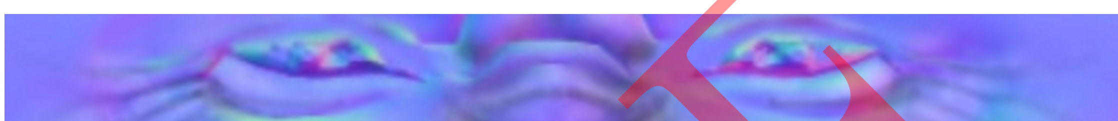
Contents

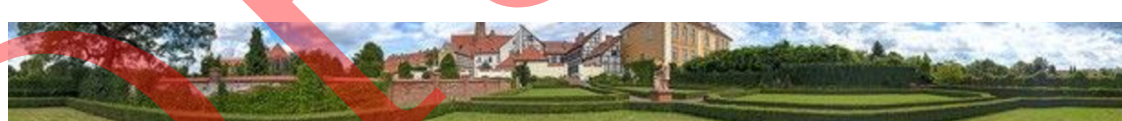


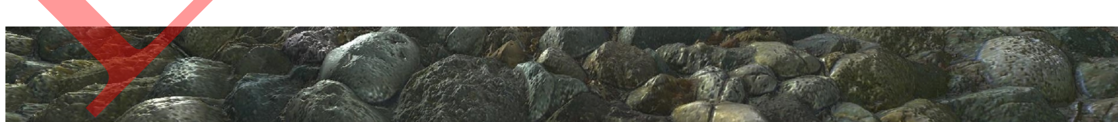


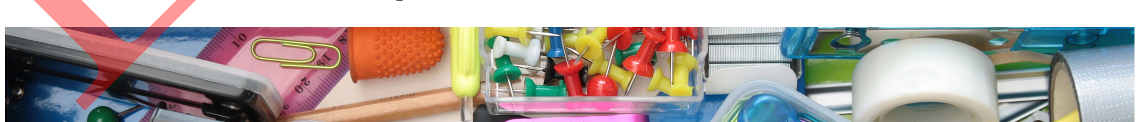


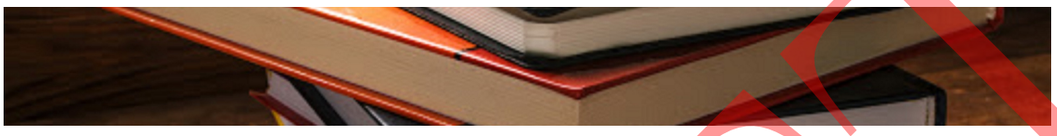


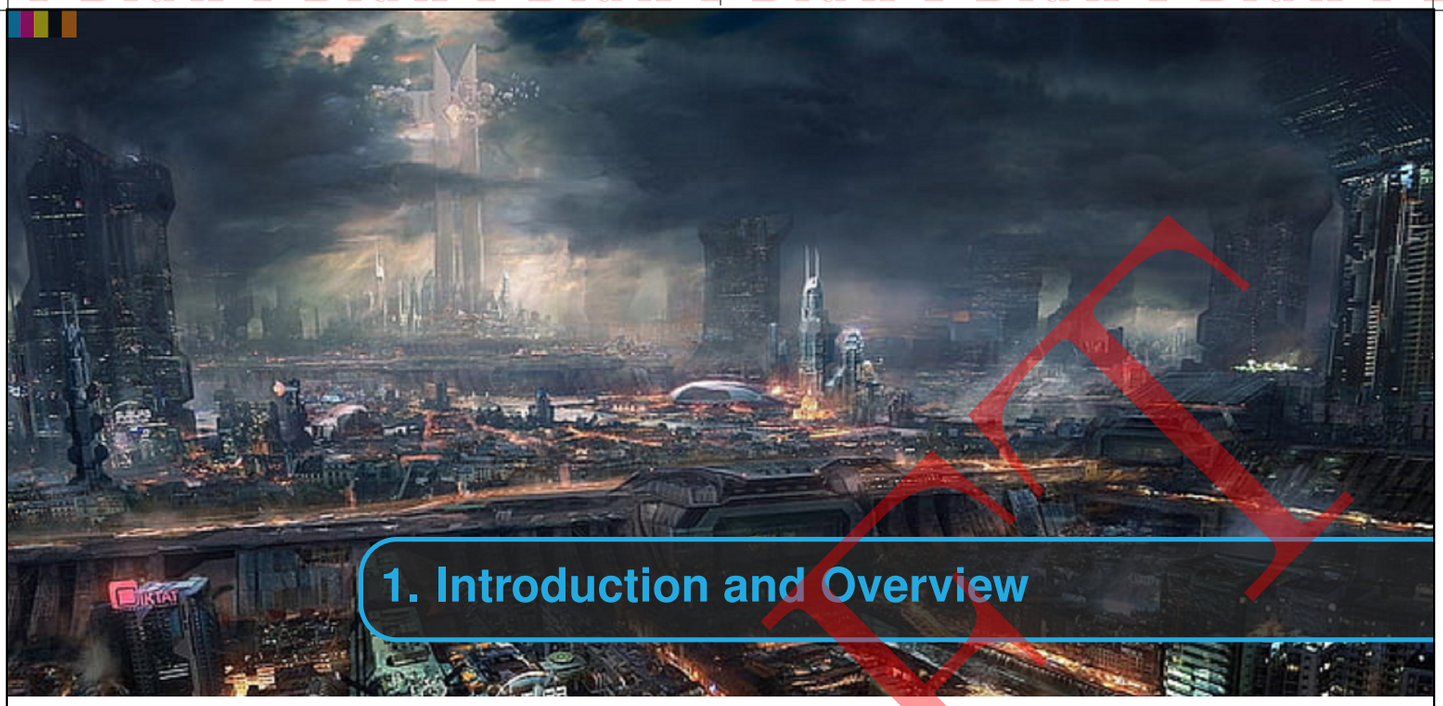












1. Introduction and Overview

1.1 Introduction

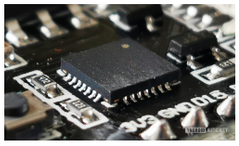


Figure 1.1: **Computer Specifications** - A dedicated GPU (or graphics processing unit) is now a necessary part of most computers (including mobile phones) designed to accelerate parallax computations, such as, in graphical applications.

WebGPU API is the successor to the WebGL API and represents the largest architectural change since the inception of browser-based GPU solutions. The primary reason for this change is the demand from the industry to provide an API that gives **more power and control to the programmer**. Vendor-specific driver implementations are often complex and impose CPU performance overheads that developers have no control over. Much of these overheads could be avoided if the API gave more control back to the developers. One example of this is the driver overhead that is present in WebGL resource management. Drivers needed to track the lifetime of every resource that is used by the rendering pipeline. Tracking of resources by the driver is often unnecessary, if it can be assumed that the programmer can perform this task with much less overhead. Providing developers with the tools to implement their own resource management takes that responsibility away from the driver implementation and often allows for performance improvements (if done correctly).

Yet with great power, comes great responsibility. It is true that this increased responsibility makes learning the WebGPU API harder than learning the WebGL API. As with all things, the first time you encounter something, it may seem daunting or too difficult to learn but if you are persistent in your desire to learn this new API, the rewards will be well worth it.

Don't worry if you are a total beginner when it comes to graphics and compute programming. This text is written with no assumptions about your current skill level and presents simplified examples to give a kick start. So even if you've never written any web GPU applications before, you should be in safe hands.

Food for Thought

Processing Power

Parallel Thinking

For more than two decades, the computer industry has been inspired and motivated by 'Moore's law' - which observed that the number of transistors on a chip double every 18 months. This observation created the anticipation that the performance of applications from one generation to the next should also double. Constant improvements in manufacturing and processor technologies has not always feed through to software applications. As new processor designs have moved away from pushing 'speed' limits, but instead towards increased transistor density which leverage parallel modalities. The WebGPU API was designed to meet this important need. It was defined and managed by the nonprofit technology consortium Khronos. In order to support future devices, WebGPU API defines a set of mechanisms that if met, the device could be seamlessly included as part of the web-browser environment.

Figure 1.2: **Done Correctly** - Taking advantage of massively parallel GPU architectures can be a big boom for performance. **Only** if the algorithm is suitable and if done properly (not every algorithm/processing project is suitable for the GPU).

WebGPU API is used to accelerate computations in web-based applications that would traditionally be handled by the CPU on the browser. Even though GPU programming has been viable for the past few decades using native applications, its applications in browsers have been limited. Since GPU programming has become a core resource in virtually every industry, the WebGPU API couldn't have come at a better time. For example, GPU programming can been used to accelerate video, digital image, and audio signal processing, statistical physics, scientific computing, medical imaging, computer vision, neural networks and deep learning, cryptography, and even intrusion detection, among many other areas.

No doubt that this is a very important text that provides students, hobbyists, academics and researchers with a better understanding of the world of GPU programming in general and the solutions provided by the WebGPU API in particular. The book is written so it fits with different learners experience levels; and can be used either as a stand alone text or as a reference guide to extend other material (lookup projects/examples).

Warning: The WebGPU API is still in the final phases of testing/deployment for modern browsers (Chrome/FireFox). For details/status updates (such as bugs and which features are available) you can check online at: **https://github.com/gpuweb/gpuweb/wiki/Implementation-Status**

Think About It!

Should you use a wrapper library instead of the WebGPU API directly?

In this day and age there are many free and commercial libraries which you can get off the shelf (e.g., Three.js). However, when you're learning computer graphics, it helps you master the topic with greater understanding if you write your own implementation (appreciate and comprehend certain principles which might not be explicitly clear if you were to use an existing library - e.g., adhoc workarounds for numerical errors and stability issues).
Of course, if you were to use a pre-existing library (mature library that has been used and tested by multiple developers/games) - then you can be sure of its stability and has been highly optimized - and may save you time and let you focus on other graphical elements (like the shader effect not the loading models and setting up boiler plate code).

However, when you are on the cutting edge - that leverage's new techniques or technologies - then you might be better off writing your own from the ground up - so you understand what's happening and why. Also if you're developing a bespoke feature which isn't supported by an existing library - you have to decide how much time you want to put into integrating this new feature into an existing library vs writing a new library/feature.
Ultimately though, you should comprehend the core mechanics - the mathematical, algorithms and graphical concepts - for example, how the graphics pipeline works and so on (vs just calling methods with no idea of what's happening and why).

Note: There are numerous good books out there, where the authors are more interested in showing you how clever they are, instead of actually trying to teach a concept. Therefore, in most cases we'll use simple examples and avoid trying to confuse you. It's the aim of this book to educate, not to impress you so much that you stop reading. Have fun and relax.

All major browser engines are working on implementing this the WebGPU specification. Visit **https://caniuse.com/webgpu** to get an overview of where all browsers and the different versions are:

1.2 Who is this book for?

This text is written for academics, creative coders, game developers, hobbyists or anyone interested in getting started with the WebGPU API. Ideally, a bit of coding experience, some basic knowledge of linear algebra and trigonometry, and a passion for learning won't harm. This text will teach you how to use and integrate the WebGPU API into your projects, improving their performance and graphical quality. Since the WebGPU

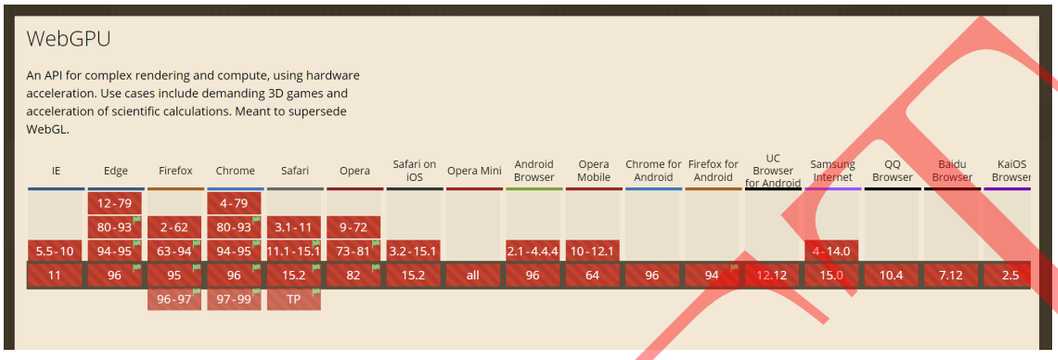


Figure 1.3: **Browser WebGPU Status** - Visually see the WebGPU status of the most popular browsers online at **https://caniuse.com/webgpu**.

API is a web-based standard which will be supported by all of the latest browsers - means you'll be able to run your projects on a variety of platforms. Also the text introduces the reader to a new shader language, the WebGPU Shader Language (WGSL) which was developed in parallel to the WebGPU API.

1.3 What does this book cover?

This book will focus on the use of WebGPU API. You'll start by learning the basic principles using minimal working examples (initializing the API). Then you'll continue to develop more complicated projects - each project extending and taking the concepts to a higher level. You'll learn the foundations of the WebGPU shading language (WGSL) and apply it to more useful scenarios such as: image processing (image operations, blurs and other effects) and simulations (water ripples). Towards the end of the book, you'll see a set of advanced techniques (deferred shading/ambient occlusion).

As you go through the material, there are interactive examples for you to play with. When you change the code, you will see the changes immediately (easy to update and run the code). The concepts can be abstract and confusing, so the interactive examples are essential to helping you learn the material. The faster you put the concepts into motion the easier the learning process will be.

1.3.1 What this book do 'NOT' cover

This book isn't about developing or learning a wrapper library/framework for the API. It's about learning the native WebGPU API through a variety of different projects - so you get a solid foundation of what the API does and how it works.

This is not a maths text. Although a number of the of algorithms and techniques are explained, if your understanding of algebra and trigonometry is weak, then you may struggle in some cases. In these cases, it's recommended that you consult one of the many texts on and around the subject (e.g., principles of matrices/linear algebra/transforms/trigonometry).

1.4 WebGPU vs WebGL

GPUs understand computational problems in terms of graphics primitives, early efforts to use GPUs as general-purpose processors required reformulating computational problems in the language of graphics cards. Fortunately, it's now much easier to do GPU-accelerated computing thanks to the new WebGPU API. This new API allow developers to ignore the language barrier that exists between the CPU and the GPU and, instead, focus on the higher-level computing concepts. Initially, WebGL was released which quickly became the dominant API for web-based GPU solutions. The API was designed to accelerate the creation of images for output to a display device which was fine at the time. The WebGL API helped developers dramatically speed up graphical applications by harnessing the power of GPUs. Of course, to use the API for non-graphical applications was a problem. Developers needed to have skills in graphical programming to manipulate and control the WebGL API to create compute solutions (e.g., storing compute data in textures, performing the calculations and then reading the result back). The WebGPU API changes all that - now, developers will be able to start creating GPU-accelerated web applications that focus purely on 'compute' problems (without any need or graphical knowledge). As technologies have changed over the decades, WebGL has tried to remain backward compatible with existing designs. The WebGPU API is a fresh design that lets developers get more access to low-level hardware/resources for improved performance/acceleration/control. So web-based applications can reach a new higher level.

Figure 1.4: **WebGPU vs WebGL** - Web-based GPU APIs.

1.5 Book layout

Most of the code in the book text consists of rather long statements (such as blocks of code). Hence, some statements may wrap around to the next line. If you have problems reading the code samples in the text, remember you can view the source code for all the sample programs and more online at: **https://webgpulab.xbdev.net**

1.6 Plan for 'Change'

Even though the WebGPU API is in the final phases of release, they are still subject to potential changes as well as specification uncertainties. You should not ignore this - just because your program works today - doesn't mean it will work tomorrow. Make sure you keep track of any changes and feed them through to your applications (regularly check and schedule time for fixes).

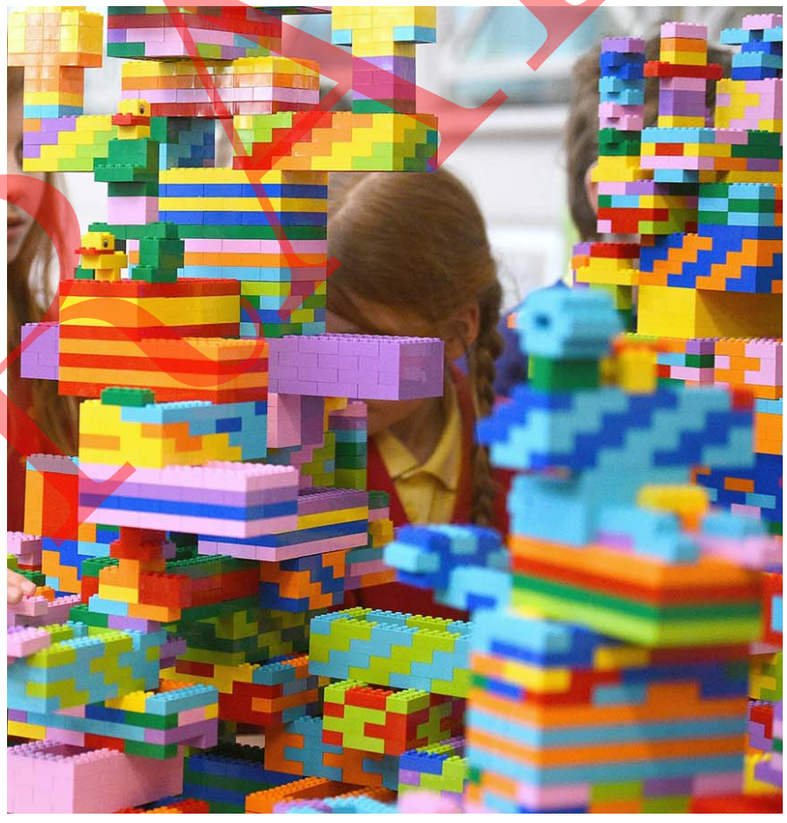


Figure 1.5: **Understand How Each Block Works/Connects** - Learning and understanding how all the API components work and operate; will hopefully, give you more freedom and control later on so that you're able to create bespoke applications that fit your needs (won't be hindered or held back by an existing framework).

1.7 Bottom Up Approach (Building Blocks)

Figure 1.6: **Bottom Up** - Nothing to everything. Starting from simple examples which are extended and expanded upon so you eventually get to a complex result (understand how and why everything fits together).

This book takes a bottom up approach to understanding the WebGPU API. Not about teaching you an existing library or framework. Instead, this text, works with very minimalistic working examples; then incrementally modifies the implementations to address limitations/problems while adding extra features and understanding.

Bottom-up approaches are always thought to be very inefficient ways to approach new subjects initially, and are often contrasted with top-down learning, which is thought to be more efficient (e.g., starting with a complete fully working program/library and take it to pieces).

The reason for this, is we believe that learning the WebGPU API should be done using a bottom-up ideology; it gets you involved with small programs from the beginning while boosting your engagement and satisfaction. Keep in mind that everyone is different; not everyone will be experts in coding and graphics. Learning this way (from basics) - will also give you an opportunity to detect pain points and plug those skills gaps (anything you're unfamiliar with). In addition, plenty of value can be derived from going over the basics. Implementing and learning smaller programs that are actually relevant and useful will also come in useful later on (when you start to develop your own applications).

At the same time, factors such as code complexity, technological advancements/updates and time are all important; but a forward-thinking approach in addition to understanding the challenges and depth of the API cannot be underestimated (all too easy hide away functionality until it comes back to haunt you later on).

1.8 Programming Language (JavaScript and WGSL)

The majority of this text evolves around practical examples (not just theoretical). Nearly all of the examples are written in mostly in JavaScript and WGSL (WebGPU shader language). It is helpful, if you're already experience with web-programming, i.e., JavaScript syntax, declaring variables, functions, lambdas and arrays; otherwise, you may struggle in some aspects. However, that doesn't mean you can't combine this text with other learning resources/books. JavaScript is one of the world's most popular programming language; also the programming language of the Web. While

JavaScript is easy to learn, it's also a language which can cause people a lot of pain and suffering; so be careful!. While this text focuses on the WebGPU API, be aware, poorly structured JavaScript (or WGSL) code - due to unfamiliarity with the language syntax/features may cause your WebGPU journey to be very bumpy. Also the programs are written for client-side applications - such as setup/data loading and so on (that is you don't need an active server to run the WebGPU programs).

> **Warning:** One of the biggest mistakes early JavaScript developers make - especially developers who have come from other programming environments (like Java or C++) are burned by misconceptions of curly brackets and variable scope (block level scope for `var` variables).

- Familiar with the JavaScript language features
- Careful about Memory Leaks (global variables stay around or any variables that are still referenced)
- Remember the Types of Equality `==` , `===` , `!=` , `!==`
- Take advantage of `"strict mode"`

When it comes to programming the GPU, you'll have to use shaders (small programs that run on the GPU). These programs are written in a shader language (few different shader languages available). However, with the introduction of the WebGPU API came a new shader language known as 'WGSL' which is short for 'WebGPU Shader Language'. The WGSL has similarities with JavaScript, such as, the syntax for declaring simple variables (`var` and `let`). Of course, as you'll learn in later chapters, don't let these similarities make you neglect learning the official WGSL syntax (otherwise, again, you'll end up in a world of pain).

1.9 Summary

Don't be ignorant to the power of the GPU!

The WebGPU API has emerged to fill a need. Over time, once the WebGPU API is accepted, it will let developers make significant advancements (performance/applications).The biggest advantage for WebGPU API is the potential for huge cross-platform access through a web browser (no plug-ins or external libraries required). The idea is that anything that has a web browser (which is just about everything) is now capable of taking full advantage of the systems resources (GPU). Unfortunately, there are still minor issues, the WebGPU API is still in the final stages of development, and there is the conundrum of support for older browsers/devices. However, there is a big push by major browsers to support the WebGPU API (which means exciting times ahead for web-applications - especially when combined with other technologies and APIs, such as, eXtended Reality

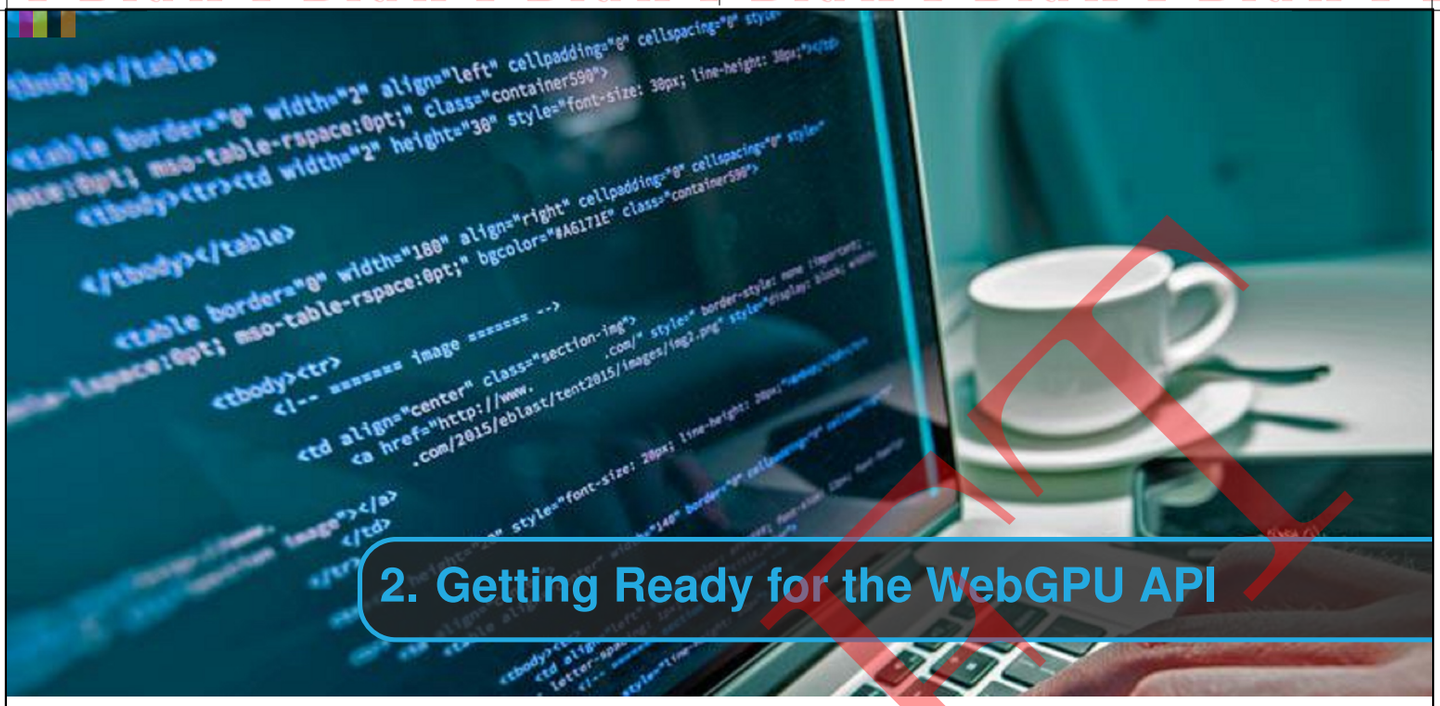
(WebXR API) and Speech-to-Text/Natural Language APIs.

1.9.1 Code Samples

Learners often rely on sample code as a crutch. The goal of code samples should be to give enough to get started but not to give away the fun of solving the problem. As a result, the sample code provided in this text is focused on the task at hand (small examples). As you work your way through the text, you're encouraged to implement good coding practices as you develop your applications using the WebGPU API. The code samples demonstrate the subject in a way to make it understandable. This often may not be the best or most efficient solution, however, there are also a large number of resources/texts online that can be combined with this text (complement your learning).

1.9.2 Exercises: Challenges

You may just want to get to coding! The "challenges" at the end of each section, will present programming challenges that allow you to apply what you have learned. They are designed to get you thinking about the application of the topics and often result in tools or sample code that can be used in later projects.



2. Getting Ready for the WebGPU API

2.1 Introduction

Figure 2.1: **Takeoff** - Launching your first program - and hopefully many more to come.

The promise behind WebGPU is an awesomely faster API that provides lower level control to the graphic resources from JavaScript. However, getting started with the WebGPU API takes a bit more work - as the API has exposed more of underlying hardware (which you have to configure and manage).

Now it is time to start your learning journey together. Your first steps introduce you to the basics of the WebGPU API (check if the WebGPU API is available, initialize a device, access resources and memory, create a simple pipeline, output some graphics to the screen). From this beginning, which uses the usual 'Hello World' type introduction (can't break away from tradition) - followed by step by step advancements that take incrementally improve and extend your knowledge. At the end of this section, you'll hopefully have the WebGPU API up and running - be able to demonstrate some core features of the API.

Want to try out WebGPU API straight away? The examples in these notes are also presented online using an interactive playground where you can write, view and experiment with existing projects immediately (**https://webgpulab.xbdev.net**).

You'll be glad to hear, that the WebGPU API is integrated in with the browser - and doesn't depend on any external resources. You should be able to setup and run a program using the WebGPU API straightaway. HOWEVER, your browser does need to support and have the WebGPU API enabled.

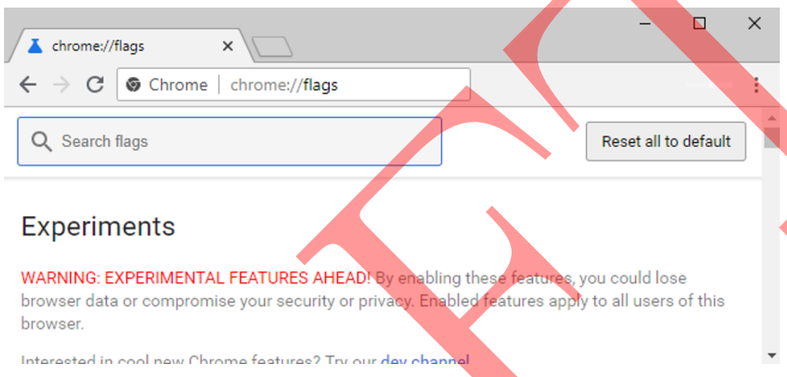


Figure 2.2: **Browser Feature Flags** - While the WebGPU API is in the final phases of development, it is disabled by default (necessary to check your browsers settings/flags) if you want to run applications that use the WebGPU API.

2.2 Checking your WebGPU API Status

Before you can start developing applications using the WebGPU API, you need to check that it's available.

```
if ("gpu" in window.navigator)
{
  console.log('Yes!!!  WebGPU is supported!');
}
else
{
  console.log('Oh no!  Either WebGPU is disable or not supported by your browser');
}
```

> **Warning:** Just because the `"gpu"` object is available in the `navigator` - this is not a guarantee that the WebGPU API is fully functional and has all of the specification features. Also it doesn't guarantee that it is enabled or has the necessary privileges to access the GPU (may fail or return null when trying to access the device).

2.3 WebGPU API Components

Before pushing further with coding examples, it's worth taking some time to quickly review some of the key components that make up the API (e.g.,

pipeline, buffers and binding groups). These are shown in Figure 2.3.

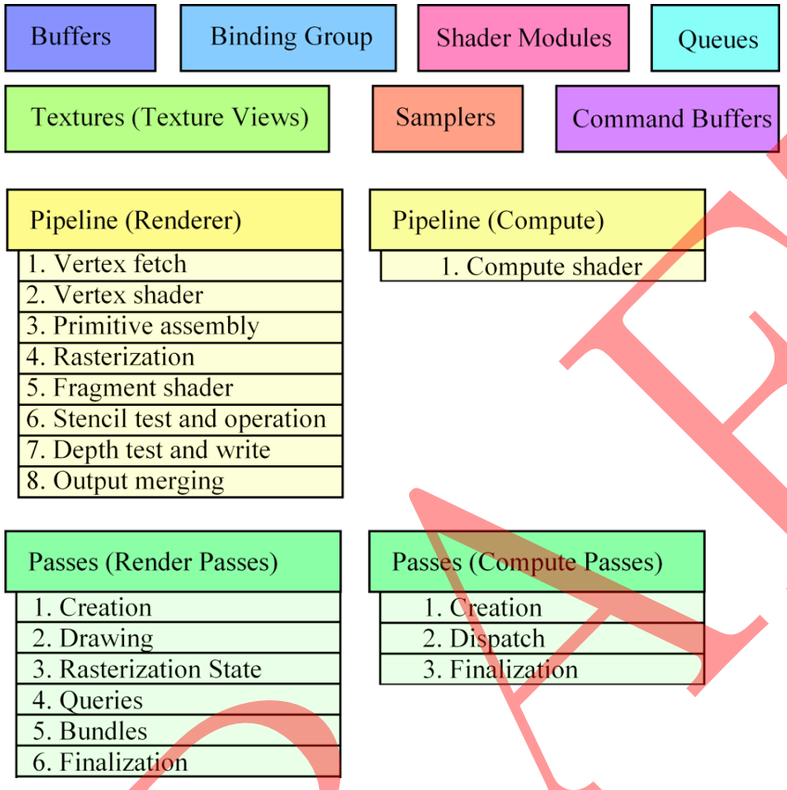


Figure 2.3: **Core Components** - A simplified visual overview of the core components within the WebGPU API.

2.3.1 Buffers

Buffers are manage blocks of memory on the GPU (used for GPU operations/shaders/pipeline). Any data, vertex data, control structures and uniforms for the shaders need to be created so that they're accessible on the GPU (i.e., `GPUBuffer`). When you create the buffer, you define its size and usages (access permissions), then you can either copy predefined data across or use it to store/receive data.

2.3.2 Binding Groups

Binding groups are the glue that connect the shaders, the buffers, the textures and the pipeline together (they ensure consistency between sizes/structures/formats).

2.3.3 Shader Modules

Shaders are written in a text based human-readable format (WGSL language). These shaders are compiled into the machine language for the GPU (shader modules manage the conversion, the details and handles to the shader binaries).

2.3.4 Queues

Queues in WebGPU are the 'execution points' for the GPUs. Every GPU has multiple queues available, and you can even use them at the same time to execute different command streams. Commands submitted to separate queues may execute at once. This is very useful if you are doing background work that doesn't exactly map to the main frame loop. You can create a queues specifically for background work and have them separated from one another (e.g., offsceen rendering or background compute simulations).

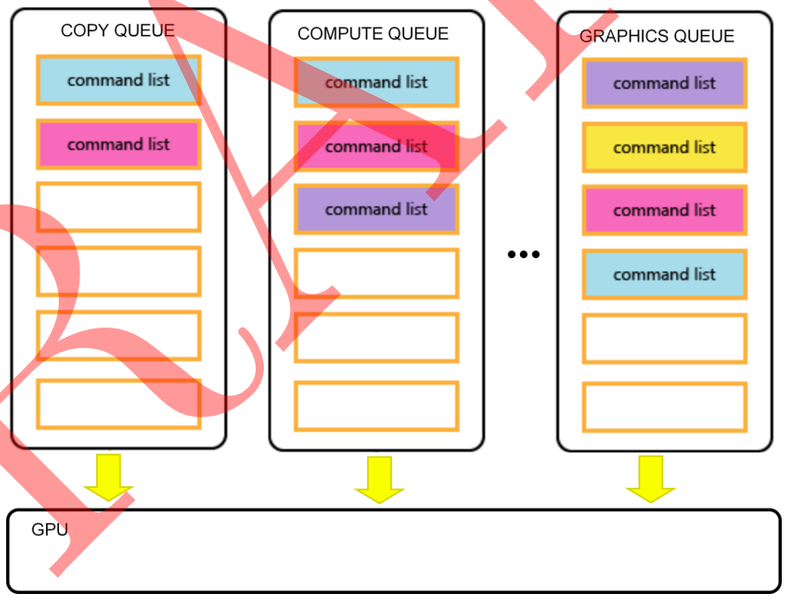


Figure 2.4: **Multiple Queues** - Distribute the workload-/commands across multiple queues with different configurations (more flexible and customizable).

2.3.5 Textures (Texture Views)

Textures come in a range of formats (1d/2d/3d/cube) and can be used in various contexts/layouts (multiple resolutions/mipmaps). Hence, textures are stored differently to raw buffer data. Texture views are a means to

manage access control to the texture data (access the texture data using a texture view).

2.3.6 Samplers

WebGPU samplers encode transformations and filtering information that can be used in a shader to interpret texture resource data. Essentially, the sampler determines how the information is interpreted from the texture. In the shader, the textures are usually accessed through samplers, which will apply filtering and transformations to compute the final color that is retrieved.

2.3.7 Command Buffers

All commands for GPU get recorded in a command buffer. All of the functions that will execute GPU work won't do anything until the command buffer is submitted to the GPU (command buffers and put into a queues for processing by the GPU).

2.3.8 Pipelines

A render pipeline (`GPURenderPipeline`) is comprised of the following render stages:

1. Vertex fetch, controlled by `GPUVertexState.buffers`
2. Vertex shader, controlled by `GPUVertexState`
3. Primitive assembly, controlled by `GPUPrimitiveState`
4. Rasterization, controlled by `GPUPrimitiveState` , `GPUDepthStencilState` , and `GPUMultisampleState`
5. Fragment shader, controlled by `GPUFragmentState`
6. Stencil test and operation, controlled by `GPUDepthStencilState`
7. Depth test and write, controlled by `GPUDepthStencilState`
8. Output merging, controlled by `GPUFragmentState.targets`

A compute pipeline (GPUComputePipeline) comprises of only a single stage:

1. Compute shader

2.3.9 Passes

The rendering (or compute operation) in takes place during a 'Pass' (either a render pass or a compute pass). The details of which are described through a 'descriptor structure' `GPURenderPassDescriptor` / `GPUComputePassDescriptor`. The descriptor to the pass details information about the bindings and the output from the shaders.

2.4 Your First WebGPU Program (Hello Compute)

The first program will be a simple compute example. You'll create an array of data, send it to the GPU, then read it back. After reading it back, you'll write it to the console window (check the values are what they should be).

Listing 2.1: Basic WebGPU program that does a compute operation. Doesn't require a graphical output (e.g., HTML CANVAS area).

```
(async () => {                                                          A
  if (!navigator.gpu) {                                                 B
    console.log(
      "WebGPU is not supported. Enable chrome://flags/#enable-unsafe-webgpu flag."
    );
    return;
  }

  const adapter = await navigator.gpu.requestAdapter();                 C
  if (!adapter) {
    console.log("Failed to get GPU adapter.");
    return;
  }
  const device = await adapter.requestDevice();                         D

  // Get a GPU buffer in a mapped state and an arrayBuffer for writing.
  const gpuWriteBuffer = device.createBuffer({                          E
    mappedAtCreation: true,
    size: 4,
    usage: GPUBufferUsage.MAP_WRITE | GPUBufferUsage.COPY_SRC
  });
  const arrayBuffer = gpuWriteBuffer.getMappedRange();

  // Write bytes to buffer.
  new Uint8Array(arrayBuffer).set([0, 1, 2, 3]);                        F

  // Unmap buffer so that it can be used later for copy.
  gpuWriteBuffer.unmap();

  // Get a GPU buffer for reading in an unmapped state.
  const gpuReadBuffer = device.createBuffer({                           G
    mappedAtCreation: false,
    size: 4,
    usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.MAP_READ
  });
```

Console output for Listing 2.1:

```
"0":0,"1":1,"2":2,"3
↪ ":3
```

```
  // Encode commands for copying buffer to buffer.
  const copyEncoder = device.createCommandEncoder();                    H

  copyEncoder.copyBufferToBuffer(                                       I
    gpuWriteBuffer,      // source buffer
    0,                   // source offset
    gpuReadBuffer,       // destination buffer
    0,                   // destination offset
    4                    // size
  );

  // Submit copy commands.
  const copyCommands = copyEncoder.finish();
  device.queue.submit([copyCommands]);                                  J

  // Read buffer.
  await gpuReadBuffer.mapAsync(GPUMapMode.READ);
  const copyArrayBuffer = gpuReadBuffer.getMappedRange();

  console.log(new Uint8Array(copyArrayBuffer));                         K
})();
```

A Certain methods of the WebGPU API take a while to complete, hence the API supports asynchronous methodologies. Methods that do not complete instantly, return a `Promise` which can be used to wait for a completion event. To manage this asynchronous behaviour and make it easier to understand use the `async` and `await` features.

B Standard WebGPU check (if the API is available).

C You get an adapter, you call `navigator.gpu.requestAdapter()`. `requestAdapter()` never fails, but may resolve to `null` if an adapter can't be found. An adapter may become unavailable; if it is unplugged from the system, disabled to save power, or marked 'stale'.

D You get a logical device by calling `adapter.requestDevice()`.

The `GPUDevice` provides APIs to create GPU objects such as buffers and textures, and execute commands on the device. WebGPU separates the concept of physical and logical devices. A physical device usually represents a single complete implementation (excluding instance-level functionality), of which there are a finite number. A logical device represents an instance of that implementation with its own state and resources independent of other logical devices. Physical devices are represented by `GPUAdapter` handles.

E Create a buffer on the GPU.

F Copy data to the buffer on the GPU.

G Create a second empty buffer on the GPU.

H Setup a command on the GPU for performing an operation.

I Add the command to copy from one buffer to the other (on the GPU)

J Issue the command to execute the stacked commands (performs the task). The `queue` allows you to send work asynchronously to the GPU. At the moment (current API version), you can only access a single default queue from a given GPUDevice (however, this is bound to change with future updates).

K Copy the data from the second buffer back to the client program and print it to the console.

The implementation might not seem very useful - but it has demonstrated that your device is available, you can allocate a block of memory on the GPU them move data around. As you'll see later, these buffers on the GPU can be passed between the compute and the graphics pipelines for storing or providing data for calculations.

Note: Even though Listing 2.1 produce any graphical output (nothing on screen), doesn't mean the concept can't be used for graphical calculations. For instance, compute shaders can be used for image processing or an offline rasterization/ray-tracing methods. As the compute configuration/buffers means the device isn't dependent on any HTML graphical output resources - the concept can be extended to include programmable shaders that allow any parallal computations to be performed on the buffers (more than just copying as shown in the example) - compute programs can be used for anything.

2.5 Your First WebGPU Program (Hello Graphics)

Figure 2.5: **Triangle** - Output for Listing 2.2.

Just for completeness, you can see a 'minimal' example of a WebGPU program that outputs something graphically (**triangle**) to the screen. While the listing only outputs a simple green triangle (Figure 2.5) the setup code is quiet long. The WebGPU API has no 'defaults' (no hiding), the setup/-configuration has to be defined by you. Hence, you need to include a simple 'shader'. You'll learn more about the WGSL shader language in detail later on - but for now, to get a basic graphics program up and running it's included here. The program simply passes the values onto the output (no complex transforms or buffers).

Listing 2.2: Sets up a basic pipline for rendering a single green triangle to the screen.

```
let canvas = document.createElement('canvas');
document.body.appendChild( canvas );
console.log( canvas );
canvas.width  = canvas.height = 512;
console.log( 'w:', canvas.width, 'h:', canvas.height );
```

A

```
let wgsltxt = `
```

```
[[stage(vertex)]]
fn main_vs([[builtin(vertex_index)]] VertexIndex : u32)
        -> [[builtin(position)]] vec4<f32>
{
    var pos : array<vec2<f32>, 3> = array<vec2<f32>, 3>(
            vec2<f32>( 0.0,  0.5),
            vec2<f32>(-0.5, -0.5),
            vec2<f32>( 0.5, -0.5));
    return vec4<f32>(pos[VertexIndex], 0.0, 1.0);
}
```

B

```
[[stage(fragment)]]
fn main_fs() -> [[location(0)]] vec4<f32>
{
    return vec4<f32>(0.0, 1.0, 0.0, 1.0);
}
```

C

```
`;

const gpu = navigator.gpu;
const adapter = await gpu.requestAdapter();
const device = await adapter.requestDevice();

// context type 'gpupresent' is deprecated. Use 'webgpu' instead.
// const ctx = canvas.getContext("gpupresent");
const ctx = canvas.getContext("webgpu");

let configuration = {
  device: device,
  format: ctx.getPreferredFormat(adapter),
  size  : { // the size of the canvas element in pixels
            width:  canvas.width,
            height: canvas.height  }
};
const res = ctx.configure( configuration  );
console.log( 'getCurrentTexture():', ctx.getCurrentTexture() );

const wgsl = device.createShaderModule({
  code: wgsltxt
});

const pipeline = device.createRenderPipeline({
  vertex  :  { module  : wgsl, entryPoint: "main_vs" },
  fragment:  { module  : wgsl, entryPoint: "main_fs",
               targets : [{ format: "bgra8unorm" }] },
  primitive: { topology: "triangle-list" }
});

let render = function () {
    const commandEncoder = device.createCommandEncoder();
    const textureView = ctx.getCurrentTexture().createView();
    const renderPassDescriptor = {
    colorAttachments: [ { view     : textureView,
                          loadValue: { r: 1, g: 1, b: 1, a: 1 },
                          storeOp  : 'store' }
    ]
  };

  const passEncoder = commandEncoder.beginRenderPass(renderPassDescriptor);
  passEncoder.setPipeline(pipeline);
  passEncoder.draw(3, 1, 0, 0);
  passEncoder.endPass();
```

```
  device.queue.submit([commandEncoder.finish()]);
  requestAnimationFrame(render);
};
requestAnimationFrame(render);
```

A In order to see what you're drawing, you'll create a `HTMLCanvasElement` and setup a Canvas Context from that canvas. A Canvas Context manages a series of textures you'll use to present your final render output to your `<canvas>` element.

B Vertex shader (written in the WGSL shader language)

C Fragment (or Pixel) shader (written in the WGSL shader language) performs computations on each 'pixel'. The example is very minimal, and simply returns a fixed color for each pixel (green).

Food for Thought

Shader Languages

WGSL (WebGPU Shader Language)

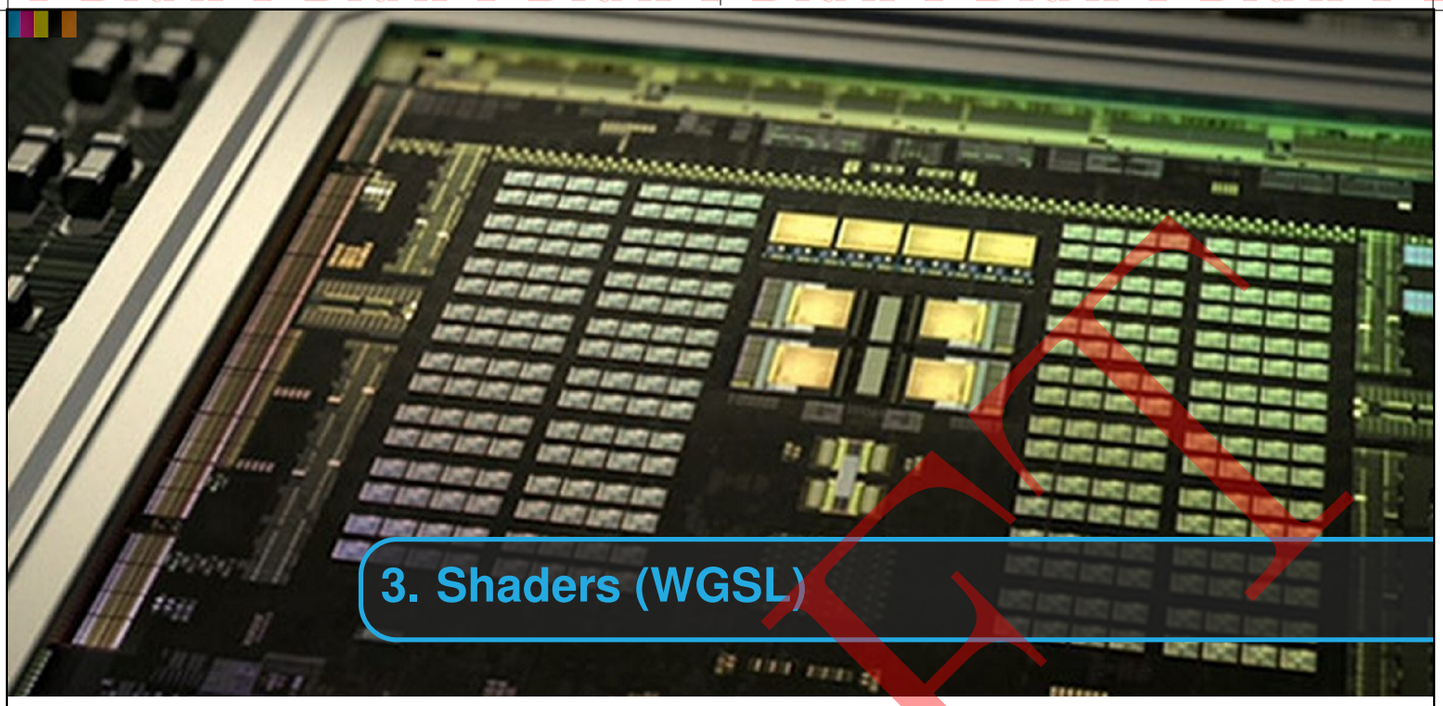
WGSL is a well-thought API that was developed in conjunction to the WebGPU specification to guarantee portability. The WGSL shader language is different from other shader languages (e.g., GLSL) for this because you don't need to translate anything at build/run-time before talking to the WebGPU API. It's also has different capabilities, such as support for atomic operations. However, the syntax can take a little getting used to if you've previously used other shading languages.

2.6 Summary

At the end of this section, you should have a minimal web application running that uses the WebGPU API. The coding examples have given you a taste of WebGPU operates, including some of the core features (e.g., GPU buffers, simple shader and pipeline). In this lesson you learned a little bit about the various components of the WebGPU API and you learned about the various stages of the pipeline. You also learned how to initialize a WebGPU application. These are important details and just the tip of the iceberg! In future sections, you will learn how to load vertices and indices into index and vertex buffers, how to write and load shader programs, perform basic lighting equations in a pixel shader, and how to perform graphical effects.

2.6.1 Exercises: Challenges

- Modify the program given in Listing 2.2, so it draws different color triangle (red, orange and blue)
- Using simple trigonometric functions (sin and cos), modify the vertex shader so the triangle is rotated
- Change the draw function so it draws 2 triangle (i.e., `passEncoder.draw(3, 1, 0, 0);` - change the 3 to a 6), update the vertex shader so it produces a 'square' instead of a single triangle
- Try modifying the graphical `topology` to different types and check the output



3. Shaders (WGSL)

3.1 Introduction

Figure 3.1: **Many Effecs** - change the shaders to create an assortment of effects/calculations (e.g., same sphere with different visual lighting calculations).

A Shader is a user-defined program designed to run on some stage of a graphics processor (e.g., vertex or fragment shader). The WebGPU Shading Language (WGSL) is the shader language for WebGPU API. As with other shader languages like GLSL, the WGSL language is not a native language understood by the GPU. WGSL is a high level language which has to be compiled to machine instructions for the GPU. The GPU only runs machine code. Even though it is currently possible to use existing OpenGL shader languages like GLSL with WebGPU; WGSL will be the way to go in JavaScript.

3.2 WGSL Syntax Basics

Overall the syntaxes are similar – WGSL is a shading language so you still talk to the GPUs in more or less the same fashion – but with some quirks especially coming from the Web flavour of GLSL. I have implemented the ability to use both WGSL and GLSL (4.5) in my WebGPU engine dgel so I'll share my findings here. Let's just do a one-to-one feature comparisons.

3.2.1 Scalar and matrix types

First off, even though the specs says that 'Plain' types in WGSL are similar to 'Plain-Old-Data' types in $C++$, you also very much feel the influence of Rust in the design.

The `bool` (true/false) behaves the same but for other scalars it seems somehow very important for you to know that they are 32-bit.

```
WGSL Scalar Types:
bool
i32
u32
f32
```

With WGSL, the component type needs to be specified in angle brackets e.g. a vector with 4 `int` elements is `vec4<i32>`. Very explicit.

If you were thinking of rejecting this new type syntax and just alias them, well sorry to disappoint but type `vec4 = vec4<f32>;` will fail as `vec4` is a reserved word.

The matrix types are also very verbose: `matNxM<f32>` with N columns and M rows (2, 3, 4 each) of floats. No shorthand here.

```
WGSL Matrix Types:
mat2x2<f32>
mat3x2<f32>
mat4x2<f32>
mat2x3<f32>
mat3x3<f32>
mat4x3<f32>
mat2x4<f32>
mat3x4<f32>
mat4x4<f32>
```

It is also not possible to generate the 'identity matrix' as easily as with `mat4(1.0)`. At least, they are still column-major.

At the time of writing, you can only pass vectors to construct the matrix (e.g., `mat2x2<f32>(vec2<f32>(1.0, 0.0), vec2<f32>(0.0,1.0));`) but specification is on the way to allow `floats` directly.

Arrays also have explicit types, `array<E,N>`, so an array of 8 vector 2 of type float will be `array<vec2<f32>, 8>`.

3.2.2 Structs

Good news everyone, apart obviously from the members declarations, the block structure is the same.

Listing 3.1: Example WGSL Structure Syntax

```
struct Light {
    //  name        type
    position    : vec3<f32>;
    color       : vec4<f32>;
    attenuation : f32;
    direction   : vec3<f32>;
    innerAngle  : f32;
    angle       : f32;
    range       : f32;
};
```

Hard to get it wrong here, just make sure to end each members line with a semicolon and not a comma.

3.2.3 Uniform buffer object

Later on, you'll use uniforms structures to pass data to the shaders (e.g., transforms or control parameters).

You need to declare a `struct` before you use the 'struct' - for instance, at the top of the shader. As shown below for a group/binding in Listing 3.2.

```
struct SystemUniform {
    // name              type
    projectionMatrix  : mat4x4<f32>;
    viewMatrix        : mat4x4<f32>;
    inverseViewMatrix : mat4x4<f32>;
    cameraPosition    : vec3<f32>;
    time              : f32;
};
[[group(0), binding(0)]] var<uniform> system: SystemUniform;
```

Listing 3.2: Example WGSL Struct Syntax

3.2.4 Functions declarations

When you declare a function in WGSL, you use the keyword `fn` followed by the function name. The arguments and return type follow the function name (separated by the `->` arrow); as shown in Example Listing 3.3.

```
fn saturate(x: f32) -> f32
{
    return clamp(x, 0.0, 1.0);
}
```

Listing 3.3: Example WGSL declaring functions.

3.2.5 Built-in

WGSL defines a number of special variables for the various shader stages. These built-in variables have special properties. They are usually for communicating with certain fixed-functionality.

WGSL name	Stage	IO
vertex_index	vertex	in
instance_index	vertex	in
position	vertex	out
position	fragment	in
front_facing	fragment	in
frag_depth	fragment	out
local_invocation_id	compute	in
local_invocation_index	compute	in
global_invocation_id	compute	in
workgroup_id	compute	in
num_workgroups	compute	in
sample_index	fragment	in
sample_mask	fragment	in
sample_mask	fragment	out

Listing 3.4: WGSL built-in defines at different stages of the shader.

Example shader setup is shown below in Listing 3.5. You need to define the main entry point function (`instance_index`) and define it in its returned value (output `struct` position).

Listing 3.5: Example of uniforms, structures, and shader entry point.

```
// UBOs
struct SystemUniform {
    projectionMatrix : mat4x4<f32>;
    viewMatrix       : mat4x4<f32>;
};
[[group(0), binding(0)]] var<uniform> system: SystemUniform;

struct MeshUniform {
    modelMatrix : array<mat4x4<f32>, 256>;
};
[[group(1), binding(0)]] var<uniform> mesh : MeshUniform;

// Output
struct Output {
    [[builtin(position)]]     position: vec4<f32>;
};

[[stage(vertex)]]
fn main(
  [[builtin(instance_index)]] instance_index : u32,
  [[location(0)]]             position       : vec3<f32>
) -> Output
{
  var output: Output;

  let modelMatrix = mesh.modelMatrix[instance_index];

  output.position = system.projectionMatrix * system.viewMatrix * modelMatrix * vec4<
      ↪ f32>(position, 1.0);

  return output;
}
```

Also note, the `discard` statement works in WGSL fragment shader. The `discard` keyword can be used within a fragment shader to abandon the operation on the current fragment. This keyword causes the fragment to be discarded and no updates to any buffers will occur. Control flow exits the shader and subsequent implicit or explicit derivatives are undefined when this exit is non-uniform.

3.2.6 No preprocessor (#define/#ifdef/#if defined())

The assumption here is that preprocessing will happen on the client side, for instance with string replacement (`#include`) or with template strings in JavaScript. On the other hand, that means less extra code in shaders so they might be more specialised and easier to read once pre-processed.

3.2.7 <f32> everywhere (32-bit)

Currently, all the data types are 32-bit (so you'll be limited to f32/i32/u32).

Arithmetic and assignments operators, l-values swizzling:

```
vec4 color  = vec4(1.0);
color.xyz   = vec3(0.1, 0.2, 0.3);
var color   = vec4<f32>(1.0);
color       = vec4<f32>(0.1, 0.2, 0.3, color.a);
```

The current specification is missing the important assignment operators (`+=` , `-=` , `*=` , `/=` , `%=` ...) as well as no increment (`++`), decrement (`--`) or exponentiation (`**`).

You'll have to make sure you don't forget this when writing your shaders, even simple operators, like `++` in loops aren't allowed.

3.2.8 Branching

The WGSL syntax has a few qwerks that you should be aware of, especially when comparing the language to JavaScript, such as, `elseif` vs `else if` . Also you have to remember, **braces are mandatory** or you'll get a syntax error if you try to exclude them.

For more details on the WGSL syntax, refer to the online documentation/specification **https://www.w3.org/TR/WGSL/#logical-builtin-functions**.

3.2.9 Braces

Bracing is mandatory, so

```
if (diff <= 0.0) return vec3(0.0); // ERROR NO BRACKETS
```

has to be expanded to:

```
if (diff <= 0.0) { // CORRECT
  return vec3<f32>(0.0);
}
```

3.2.10 Function overloading

Function overloading is a very common operation - however, it's very limited in WGSL. Following examples are not possible with the WGSL language:

```
fn toLinear(v: f32) -> f32 {
  return pow(v, GAMMA);
}

fn toLinear(v: vec2<f32>) -> vec2<f32> {
  return pow(v, vec2<f32>(GAMMA));
}

fn toLinear(v: vec3<f32>) -> vec3<f32> {
  return pow(v, vec3<f32>(GAMMA));
}

fn toLinear(v: vec4<f32>) -> vec4<f32> {
  return vec4<f32>(toLinear(v.rgb), v.a);
}
```

3.3 Look at WGSL Shader

Let's look at a simple WGSL fragment shader which tints the texture output; basically multiplies the output texture color by a constant.

```
[[binding(0), group(0)]] var samplerFront : sampler;
[[binding(1), group(0)]] var textureFront : texture_2d<f32>;

[[block]] struct ShaderParams {
        tintColor : vec3<f32>;
};
[[binding(2), group(0)]] var<uniform> shaderParams : ShaderParams;

struct FragmentInput {
        [[location(0)]] fragUV : vec2<f32>;
        [[location(1)]] fragColor : vec4<f32>;
};

struct FragmentOutput {
        [[location(0)]] color : vec4<f32>;
};

[[stage(fragment)]]
fn main(input : FragmentInput) -> FragmentOutput {
        var front : vec4<f32> = textureSample(textureFront, samplerFront, input.
    ↪ fragUV);
        var output : FragmentOutput;
        output.color = front * vec4<f32>(shaderParams.tintColor, 1.0);
        return output;
}
```

The WGSL syntax can make the shader seem verbose and long. However,

the added information is to make the code a lot less ambiguous - almost everything is spelled out explicitly. Lots of details have to be hard-coded in, like the specific location of inputs and outputs. Even a small shader has a few structs.

3.4 Attributes

Lots of things are marked up with attributes in `[[` and `]]`, such as `[[binding(1), group(0)]]`. These imbue otherwise normal variables and functions with special meanings, such as in this example, precisely which binding slot in which bind group a texture corresponds to (which corresponds to WebGPU API calls). WGSL's attributes are pretty exhaustive and can be used for everything from describing shader stages to the precise binary layout of a struct.

There's no referring to things by their name in WebGPU. You have to set it all out in the shader. This can seem like a downside, as you don't really want to have to hard-code lots of details like binding slots in shaders across a small ecosystem of third-party shaders, and WGSL does not provide a preprocessor or even any good rules about using constants instead of literals in attributes.

3.4.1 Variable declarations (var/let)

WGSL has a different syntax for variable declarations based on `var` with explicit types.

Instead of using explicit type defines for your variable qualifiers, you use `var` and `let`. Similar to JavaScript, so there's a `const` too right? No. It is only a Reserved Words for now. If you are looking for immutability, go for `let`:

```
let GAMMA: f32 = 2.2;
```

One of the goals with `var name: type` was apparently to make it closer to TypeScript. After some complaints, `type` inference has been added for the sake of conciseness and readability so you can write:

```
var position = vec2<f32>(0.0, 0.0);
```

as well as:

```
var position: vec2<f32> = vec2<f32>(0.0, 0.0);
```

Listing 3.6: Declaring a variable in WGSL shader (explicit type details).

```
var color : vec4<f32>;
```

Struct members and function parameters use a similar syntax but omit `var`.

WGSL has no precision specifiers like `lowp`. You have to give everything a specific type like `f32` (a 32-bit float). Currently there's no `f16` type either, but that should come as an extension in future - until then there's a whole lot of `f32` going on.

WGSL does support automatic type deduction, which can save a lot of typing. So if you initialise a variable to something, you don't have to specify the type, and it will be taken from the thing assigned.

Listing 3.7: Both ways are valid for declaring a variable - however, you can explicitly include the 'type' with the variable name during the declaration.

```
// This way specifies the type twice
var color : vec4<f32> = vec4<f32>(1.0, 0.0, 0.0, 1.0);

// The variable type can be omitted though
var color = vec4<f32>(1.0, 0.0, 0.0, 1.0);
```

One seemingly cruel syntax choice is that coming from JavaScript, `var` in WGSL means `let` in JS (i.e. reassignable), and `let` in WGSL means `const` in JS (i.e. not reassignable), and in JS `var` is the old thing you're not meant to use any more. The rationale for this is based on the potential wider appeal of WGSL outside the web. It may feel weird at first, but in the end you'll get used to the syntax.

3.5 Structs

In WGSL `structs` are used to represent uniform buffers as well as shader inputs and outputs. They have a specific binary layout which your **JS code will need to match** when updating them. If these are programmatically determined, such as loaded from a file, you'll also need to wrap your head around the `struct` alignment rules in the WGSL specification. You can also add attributes to explicitly place everything.

More uniquely `structs` are used for both shader inputs and outputs. This actually makes sense and is a nice way to clearly **define inputs and outputs**, with the main function accepting an input `struct`, returning an output `struct`, and all members of the `struct` providing annotations specifying their location. If there's only one input or output using a `struct` is optional, however, it's usually best to be consistent and always use `structs`.

3.6 Function Syntax

The WGSL function syntax is actually pretty nice. It's a good example of how WGSL departs from the C-like syntax and goes with something that looks a lot more like Rust. A simple add function looks like this:

```
fn add(a : f32, b : f32) -> f32
{
	return a + b;
}
```

This also applies to the shader main function, plus annotations, and using the special input and output `structs` as parameter and return value.

3.7 Texture Sampling

WGSL takes a modern approach to textures and samplers. Textures and samplers are different and both have to be specified. You just use `textureSample` which derives the type from the `texture`, which has the type `texture_2d<f32>`.

3.8 WGSL Capabilities

WGSL is a really nice language with lots of juicy little features. For example you can use the `textureDimensions` built-in to get the size of a texture in a shader. If you can get a WebGPU context you get all these capabilities as a baseline. The WebGPU specification mandates minimum capabilities, such as being able to load 2D textures sized at least 8192x8192, which you can rely on unconditionally in WebGPU code.

3.9 Ternary Operator

GLSL supports the ternary `?:` operator. WGSL does not support this, but provides the built-in function select(falseValue, trueValue, condition) which does much the same thing (although mind that parameter order!). It also provides vector overloads, which is a nice benefit.

3.10 If/Else

In WGSL braces around `if` are **mandatory**. As a result you can't write the usual C-style `else if`, as it would need to be `else if with curly brackets`. To avoid this WGSL provides a special `elseif` keyword.

```
// Sample WGSL if/elseif/else
if (colorDistance == blackDistance)
{
        finalColor = colorBlack;
}
elseif (colorDistance == whiteDistance)
{
        finalColor = colorWhite;
}
else
{
        finalColor = colorMagenta;
}
```

It's easy to forget that the **braces are mandatory**, leave them out, and get a parse error. It's another way WGSL code tends to end up verbose. But it's not the end of the world.

3.11 No Arithmetic Assignment or Increment

Arithmetic assignment operators like `+=` are not currently supported in WGSL. You'll end up with a lot of code like `n = n * 2`.

Further, there are no increment `++` or decrement `--` operators. The rationale again appears to follow Rust. Given the missing arithmetic assignment operators, it feels a bit silly to have to write `i = i + 1` in a for loop, but assuming we get arithmetic assignment then that can at least become `i += 1`.

3.12 Assigning to Vector Components

WGSL doesn't yet have what's technically known as "l-value swizzling". In practice this means you can't assign to just a few components of a vector - you have to replace the entire vector. For example:

```
// Declare a vector with 4 components
var color : vec4<f32>;
```

```
// ...

// Not yet supported: assigning to just RGB components
color.rgb = someVec3;

// Workaround: assign a whole new vec4<f32>
color = vec4<f32>(someVec3, color.a);
```

Presumably support for this will be added later. Fortunately you can combine components in vector constructors, such as `vec4<f32>(someVec2, z, w)`, as well as repeating components, such as `vec4<f32>(f)` being equivalent to `vec4<f32>(f, f, f, f)`.

3.13 Limited Vector/Scalar Overloading

Limited builtin overloading for variable types in WGSL. For example addition `+` can be used to add `vec3 + float` and return a `vec3` with the `float` added to every component. However, the less-than `<` operator only accepts vectors with the same number of components on both sides. So if you write `someVec3 < someFloat`, you'll get a syntax error and have to change it to `someVec3 < vec3<f32>(someFloat)`.

Similarly several built-in functions currently require the same vector types for all parameters, such as `pow(a, b)` and `clamp(x, a, b)`. So in WGSL to clamp a value you'd have to write `clamp(someVec2, vec2<f32>(0.0), vec2<f32>(1.0))`.

3.14 WGSL vs GLSL

WGSL doesn't aim to be compatible with other shader languges, such as, GLSL (OpenGL Shader Language). The syntax departure is good evidence of how this was a clean-slate redesign of a modern shader language. So don't assume the way something worked in GLSL will transfer across to WGSL.

Two minor examples of this are the WGSL `%` operator works slightly differently to the GLSL mod function (one uses trunc, the other floor); and `atan(y, x)` in GLSL is called `atan2(y, x)` in WGSL. There are probably several more examples. They're generally easy to work around if you refer to both specifications to see how each work.

3.15 Random Pixel Colors (Chaos)

Modify the basic triangle example in Listing 2.2, to use the WGSL shader code below in Listing 3.15. The extended example adds extra functionality to the shader (JavaScript code remains same). The modified shader passes the position information from the vertex shader to the fragment shader using a structure. You add a `random` function which calculates a random value based on the position. This random value is used for the rgb (red, green, blue) output pixel color.

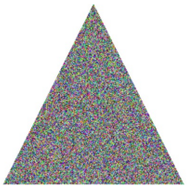


Figure 3.2: **Random Pixel Colors** - The output for the shader in Listing 3.15.

```
struct vsout {
    [[builtin(position)]] Position: vec4<f32>;
    [[location(0)]]       p       : vec3<f32>;
};

[[stage(vertex)]]
fn main_vs([[builtin(vertex_index)]] VertexIndex : u32) -> vsout
{
    var pos = array<vec2<f32>, 3>(
                vec2<f32>( 0.0,  0.5),
                vec2<f32>(-0.5, -0.5),
                vec2<f32>( 0.5, -0.5));

    var ret: vsout;
    ret.Position = vec4<f32>(pos[VertexIndex], 0.0, 1.0);
    ret.p        = vec3<f32>(pos[VertexIndex], 0.0);
    return ret;
}

fn random(st:vec2<f32>) -> f32
{
    return fract(sin(dot(st.xy, vec2<f32>(12.9898,78.233)))*43758.5453123);
}

[[stage(fragment)]]
fn main_fs([[location(0)]] p : vec3<f32>) -> [[location(0)]] vec4<f32>
{
    var r = random( p.xy );
    var g = random( p.xy*2.0 );
    var b = random( p.xy*4.0 );
    return vec4<f32>(r, g, b, 1.0);
}
```

3.16 Summary

WGSL has a great syntax, and has lots to offer once you get used to the syntax. It is powerful but also much more verbose, with every last detail of your shader having to be spelled out. You could say the WGSL "explicit is better than implicit" design principle that WGSl follows is a good idea. WGSL code is always clear as you know exactly what it is doing, because you have to specify it yourself to every last detail.

The main downside of WGSL is basically that it's a young technology. It's