LASZLO SZECSI

COMPUTER GRAPHICS WITH WEBGL.

BME/AIT

Contents

List of Figures

List of Tables

Introduction

Images in the real world are created in various ways. They might capture scenes, and reproduce their visual impression to some extent—as by a photograph. Or, the image can be a product of abstraction, already interpreting reality and presenting it in an illustrative way—like an oil painting or a blueprint.

In computer graphics we render virtual worlds by making a picture of them and presenting their image to the user. The virtual world is stored in the computer memory. The virtual world model can be the result of an interactive modeling process (i.e. artistic creation), simulation (i.e. procedural creation following physical laws or invented rules), measurement (i.e. capture of real world geometries and motions), or any combination thereof.

Rendering can be regarded as an abstract mapping from the virtual world model to the brightness and color values of the computer screen. There are infinite number of possibilities to define this mapping. If we wish to have images that look like real images, we should simulate the image creation process of the real world. For example, we can simulate light transport (i.e. optics), or manual drawing.

All numbers describing the virtual world may change in time. Therefore, they should be updated and a new image should be rendered. If the image sequence is fast enough, we perceive continuous motion, or animation.

To let the user influence the virtual world, a graphics application handles the user input. This input can be part of the modeling process (e.g. in a vector graphics editor program), or can influence the animation process. For example, in a computer game, to provide believable animation, physical simulation can be used, and user input influences that by applying appropriate force (e.g. throttle up). After changes to the virtual world, it has to be immediately rendered, presenting the resulting image of the updated world to the user. This way, the user immerses into the virtual world, i.e. she feels that she is promptly informed about its current state.

The process from the input to the virtual world is called the input pipeline. Similarly, the process mapping the virtual world to the

screen is the output pipeline.

To define a virtual world, we first set up a coordinate system and place objects, lights and the camera with respect to this system. To prepare for rendering, the material of the object should also be known. Having set the lights, camera, and objects with known material parameters, an image can be rendered. All parameters may change in time, so at short intervals they should be updated and a new image should be rendered. If the image sequence is fast enough, we perceive continuous motion or animation.

If we want to create photo-like images, we should simulate light transport and provide the user with the illusion that he watches the real world and not a computer screen.

If we could guarantee that the human eye gets the same photons (i.e. the same amount for any wavelength) from the pixel as it got from the real world at identical directions, then it would not be possible for the user to distinguish between the computer monitor and the real world, since identical photons result in an identical color sensation. Therefore, in computer graphics, we should compute the number and wavelength of photons (i.e. the power spectrum of the light) that would enter the eye from the direction of each pixel. Then the display should be controlled to emit similar photons. Fortunately, we do not have to emit exactly the same spectrum since the human eye is very bad in measuring a spectrum. In fact, the illusion of most of the spectra can be provided by carefully selected red, green and blue intensities. So having calculated the spectrum, we convert it to an equivalent red/green/blue intensity triplet and get the monitor to emit it.

The calculation of the light spectrum requires the solution of the photon transfer or the transfer of electromagnetic waves.

The display hardware controls the red, green, blue intensities according to the content of a memory where each pixel is associated with certain number of bits. This is the frame buffer memory, which is the digital representation of the display image. The output of the rendering process is this memory. In old systems, this memory was directly written by the CPU and our algorithm computed the pixel colors and placed them here. However, due to speed requirements, nowadays a special hardware, the GPU, supports the rendering process and our CPU algorithm only feeds this hardware. The output of the GPU is the frame buffer memory.

In this book, our graphics application are embedded in web pages, and displayed by the browser. We interact with the display hardware, the windowing system, and the user input devices—like the keyboard or the mouse—through the web programming API of the browser. We control the GPU through a specific part of this API

called WebGL.

JavaScript

OpenGL (Open Graphics Library) is a highly standardized, cross-platform library specification for interactive rendering, and the most widely supported interface for programming graphics hardware (GPUs) in graphics applications. It has several versions and implementations on numerous platforms. The OpenGL ES variants are somewhat limited and streamlined standards, with ES standing for Embedded Systems. WebGL 1.0 and WebGL 2.0 are JavaScript APIs implementing the OpenGL ES 2.0 and OpenGL ES 3.0 standards, respectively. JavaScript is a programming language prominently used for programs embedded in web pages, executed by browsers. All popular browsers support both JavaScript and WebGL. In this book, we will use the JavaScript version officially known as ECMAScript 6 (ES6) or ECMAScript 2015, and WebGL 2.0 for graphics.

JavaScript was initially designed for simple tasks implemented by non-programmers. It was a basic concept that if a valid interpretation for the code can be found, it just should be done[1], without burdening the developer with type matching issues or variable scope. This approach makes JavaScript pretty easy to pick up if you already know a bit of C or Java. The syntax is similar, your first guess at how to write something like a loop, a conditional, or a math operation usually works—and if you need a code snippet for something like finding and element in an array or calling a function for all properties of an object, a quick internet search gives you the answer. In this chapter, we are not going to detail JavaScipt programming, as there are ample online and offline resources for that. However, note that there are also a lot of aspects in web programming, and widespread JavaScript libraries like jQuery or Underscore.js, that we do not use. JavaScript and WebGL, for us, are vehicles for programming the GPU, and here we highlight only the essential programming tools we need for that.

The downside to JavaScript's *anything goes* approach is that it is easy to make mistakes, i.e. write code that does not actually do what our intent was, even if it appears to be. A missing semicolon, a wrongly placed curly brace, a mistyped variable name may cause disproportionate amount of headaches. There is a great deal of best

[1] JS is forgiving:

- Compare a string and an integer? No problem, the string shall be parsed.
- Set a variable never before seen? No problem, let's make that a global variable accessible anywhere.
- Access a variable only defined later? No problem, let's move its definition to the start of the function.
- Semicolon omitted at line's end? No problem, let's add it.

practices designed to avoid these pitfalls. *Linters* are tools that check your code and warn you about potential hazards and remind you to follow certain guidelines. These are often embedded into text editors. Sublime Text, for instance, has the SublimeLinter plugin, for which JS support can be added by installing SublimeLinter-jshint. It is also incredibly useful to have syntax highlighting and automatic code formatting, where the JSFormat, GLSL, and SublimeCodeIntel packages help.

We avoid using global variables. Local variables are defined with the **const** or **let** keywords, with no type specification necessary.

```
let myVariable = 1;
const myString = "Hello AIT!";
```

These variables are scoped to a code block—that is, the identifier is valid from its place of definition to the closing curly brace of the tightest pair of curly braces enclosing the definition. **const** means we cannot assign a different value, while **let** allows this. We should use **const** whenever possible—which is, maybe surprisingly, almost always—to avoid mistakenly changing a variable we did not intend to.

Variables may have number, string, boolean, or null values. A non-existent variable is undefined. Most importantly, though, variables may refer to *objects*. Objects in JS are bags of *properties*. Every propetry has a string name, and a value. Thus, properties are like variables, but contained in objects. We may create an object using an *object literal*.

```
const emptyObject =
  {};
const myObject =
  {a:1, b:"foo"};
```

The first variable definition creates an empty object (without properties) referred to by variable emptyObject. The latter creates an object, referred to by variable myObject, with two properties. We may refer to those properties as myObject.a and myObject.b. It is perfectly valid to add or remove properties at any time to or from any individual object.

```
emptyObject.p =
  {desc:"new property"};
myObject.c =
  "hey I am new here";
delete myObject.b;
```

Note that even though myObject is **const**, we can manipulate the properties freely. It is the binding of the variable to the object that cannot be changed. myObject = `"let␣this␣be␣string"` would not work. But the properties of the object are free game. That is the reason that **const** should be used over **let**. It does not prevent us

from setting the object properties, but it makes sure we do not assign something completely different to a variable we expect to refer to an object.

There is another syntax for accessing properties. myObject["a"] is identical to myObject.a. However, this syntax allows the use of property names stored in strings, as opposed to them being known verbatim when writing the code.

```
const propertyName = "a";
myObject[propertyName];
```

A special type of objects is *arrays*. They have a length property. Otherwise, property names of elements are contiguous numbers starting from zero. Special operations like push to add elements, or map to process all elements are available.

```
const myArray =
  ["rook", "knight", "bishop"];
myArray.push("king", "queen");
```

Note again, that even though myArray is **const**, we can manipulate the array elements freely.

Another, even more special type of objects is *functions*. JS is a functional programming language, making functions the ubiquitous building blocks of our code. As all functions are objects, they can be assigned to variables or properties, or passed around as parameters, freely. This allows enormously powerful programming practices (similar to C++ lambdas and Java closures) with simple syntax. This is one way to define a function:

```
const myFunction =
  function(param1, param2){
    return param1 + param2;
  };
```

Note this is just a variable assignment like any before. The variable just refers to this function. The function itself does not have a name, but the variable does. As it is **const**, the name always refers to the same function. However, we can assign the same value to another variable.

```
const theSameFunction =
  myFunction;
```

Calling the function uses the usual syntax

```
const sum = myFunction(2, 3);
```

Note that there is no checking for the type or number of parameters. Additional parameters are ignored, missing parameters are undefined, and if the function tries some operation that cannot be performed on the supported parameters, that will cause an error inside the function. Compared to strongly typed languages, this lack of

type checks removes some guardrails that help the programmer, and make developers rely on debugging runtime issues instead of weeding out compiler errors. However, there is less redundancy and more flexibility, which allows for fast development as long as the interfaces remain clear and simple.

Functions can be set as object properties.

```
const myObject = {
  f : function(a){
    return a*2;
  }
};
```

Where calling the function can happen as

```
const dub = myObject.f(4);
```

Almost always, we want these function properties, or *methods*, to access other properites of the same object.

```
const myMultiplier = {
  factor:3,
  f : function(a){
    return a*this.factor;
  }
};
```

Where **this** is supposed to refer to the object that has the method as its property. The **this** reference is automatically bound is such a way, if the method is invoked with the typical syntax.

```
myMultiplier.factor = 5;
const result =
  myMultiplier.f(3); // = 15
```

Note that **this** is not bound (and refers to a global object called window) if the same function is called, but not as a method.

```
const extractedFunc =
  myMultiplier.f;
const result =
  extractedFunc(3); // FAILS
```

Often, we want to create a lot of similar objects. Ones that have properties and methods with the same names. It makes a lot of sense to write a function that creates such an object. Such a function is known as a *constructor*, and it has some special syntax and capabilities in JS. Objects may also have *prototypes*, which allows, among other features, several objects to share methods. In this book, however, we do not dwelve into these underlying mechanisms of contructor functions and prototypes, but instead rely on the *class* syntax, that should be more familiar to C++ or Java programmers.

```
class Student {
  constructor(name, grade){
    this.name = name;
    this.grade =
```

```
      grade || "A++";
  }
}
const student =
 new Student("Joshua Smith");
```

Note the default parameter here: if grade is undefined (e.g. the constructor is called with only one parameter), **this**.grade takes the value of `"A++"`. Creating an instance with **new** Student invokes the constructor: an empty object is created, and the constuctor is called, with **this** referring to that newly created object. It is the constructor's task to populate the object with properties. If we create multiple objects (Student *instances*) with the same constructor, they will, at least initially, have the same properties, possibly with different values.

It is absolutely possible to add methods in the same way. However, we most often want the method properties to have the same value, and it makes little sense to create a new function for every instance we create. Using the class syntax, such methods shared by all created instances can be defined like the constructor, but with different names. A vector class could be written as follows:

```
class Vec2 {
  constructor(u, v){
    this.x = u.x || u || 0;
    this.y = u.y || v || 0;
  }
  add(u, v) {
    this.x += u.x || u || 0;
    this.y += u.y || v || 0;
  }
}
```

And it can be used as:

```
const v1 = new Vec2(1, 2);
const v2 = new Vec2(v);
v1.add(v2);
```

Methods are typically created described above. However, functions that are not methods are ubiquitous in JavaScript code. As we avoid globals, these functions can only be defined as local variables in other functions, very often in methods of classes.

```
class Student {
  ...
  getSecretName() {
    this.name.replace(
      /[aeiou]/gi,
      function(match) {
        return match +
               "r" +
               match;
      }
    );
  }
}
```

Here, the replace method of the string expects two parameters. The first is a *regular expression*. Here we pass one that matches all wovels. The second is a function that determines what to replace them with. It is called by the replace method as many times as there are matches in the string. Here, an r is added and the wovel is repeated, turning `"John␣Doe"` into `"Jorohn␣Doroere"`. The point is that a function is defined in place, and instantly passed on as a parameter to a different function call.

In order to make our name cypher more customizable, we could make the extra character to be inserted a parameter of the method.

```
class Student {
  ...
  getSecretName(extraChar) {
    return this.name.replace(
      /[aeiou]/gi,
      function(match) {
        return match +
               extraChar +
               match;
      }
    );
  }
}
```

The above code works. That may be surprising, as we made quite a leap of faith by just using extraChar in the locally defined function. It is a variable of the enclosing scope, and has not been explicitly passed to the function. Why this still works is because JS supplies locally defined functions with *closures*. Every locally defined function comes with a set of invisible variables: those of the enclosing scope at the time of the function definition. This means that copies of the local variables from the enclosing scope are saved whenever a function is defined locally, for the function to operate on later, when invoked. Just like a piece of amber would keep animals of old, so does a locally defined function remember its defining scope—hence the name closure. Practically, this just means we are free to use the outside variables in our locally defined functions, intuitively. Note that properties that reference objects are copied as well, but new objects are not created. Closures keep referencing the same objects, and their properties may very well change before the closure function is called.

There is one keyword that we would often like to use in closures, but cannot: **this**. The closure function is not called as a method of any object. Therefore, **this** does not refer to to same object as **this** in the enclosing scope does. A simple and widespread workaround is to define a local variable in the enclosing scope, and set it to **this**. That variable is captured in the closure, and can be used inside the closure function. This practice, however, introduces a bit of extra code, and

does not read very well. A solution is to use *arrow functions*. The following definitions are almost identical.

```
const f =
  function(a, b){
    return a+b;
  };
```

```
const f =
  (a, b) => {return a+b;};
```

The only difference is that instead of the **function** keyword appearing before the parameter list, we type an arrow between the parameter list and the function body. Both define a function. One important difference is, though, that arrow functions do not bind **this** — meaning that it keeps referring to the same object as in the enclosing scope. In practice, we just always use arrow functions for closures, and we can use the **this** keyword inside them intuitively.

Web programming

WebGL is a JavaScript API. We need a HTML5 *canvas* element that appears on a web page, displayed by the browser. For that canvas, we can obtain a *WebGL context* object, which has numerous methods we can use to set up the GPU to render images displayed on the canvas. These methods are, for the most part, almost identical to functions in other OpenGL library implementations, no matter the programming language. In this chapter, we discuss how to set up a web page and obtain the WebGL context, without going into any graphics programming. Thus, here we discuss the minimal web programming framework we need. Once this is done, we may never need to touch this part of the code again. However, handling interactions like mouse clicks and key presses do require some understanding of how the browser and the web page are represented in our code.

The webpage is defined in an HTML file. The *head* part defines some display options for the browser. An important setting here is that our webpage extends to fill the browser window. The display style of webpage elements is defined in a CSS file. The *body* part contains the tags that describe the hierarchy of webpage *elements*. The most important for us is the canvas, but it is useful to have another element overlaid on it for text rendering. This is a div element, the actual HTML content of which we can change from JavaScript, displaying arbitrary text. JavaScript files are also embedded in the HTML file. These scripts are executed as the HTML file is processed, but as we will see, they typically only define constructors and methods, and we call those functions only after the webpage has loaded completely.

```
<!DOCTYPE html>
<html>
<head>
  <title>AIT Graphics</title>
  <meta name="viewport"
    content="width=device-width, initial-scale=1" />
  <meta http-equiv="Content-Type"
    content="text/html; charset=utf-8" />
  <link rel="stylesheet" type="text/css"
    href="css/style.css" />
</head>
```

```
<body>
  <div id="container">
    <canvas id="canvas"></canvas>
    <div id="overlay">Overlaid text.</div>
  </div>
  <script src="js/keycodes.js"></script>
  <script src="js/Shader.js"></script>
    <script src="js/shaders/idle-vs.glsl"></script>
    <script src="js/shaders/solid-fs.glsl"></script>
  <script src="js/Program.js"></script>
  <script src="js/QuadGeometry.js"></script>
  <script src="js/Scene.js"></script>
  <script src="js/App.js"></script>
</body>
</html>
```

The CSS file makes sure that the canvas fills the entire browser window by setting element widths and heights to 100. Absolute positioning is used to overlay the div element on the canvas.

```
* {
  margin: 0;
  padding: 0;
}
#container {
  position: absolute;
  width: 100%;
  height: 100%;
  top: 0;
  left: 0;
  bottom: 0;
  right: 0;
  overflow: hidden;
}
#overlay {
  position: absolute;
  left: 10px;
  top: 10px;
  z-index: 20;
}
#canvas {
  background-image:  url("bg.png");
  position: absolute;
  width: 100%;
  height: 100%;
  top: 0;
  left: 0;
  bottom: 0;
  right: 0;
  z-index: 10;
  overflow: hidden;
}
```

Our scripts interact with the webpage through the *Document Object Model* or *DOM*. This means that HTML elements like the canvas or the overlay div appear as objects in JS, and we can manipulate them through methods. We only use an extremely small sliver of the extensive web programming arsenal available in JS.

The root element of the hierarchy of DOM objects is called document.

The method getElementByID can be used to obtain a certain element, like the canvas. The other global object of use is called window. The document and window are closely related, and responsibilities are divided between them – e.g. we need window if we wish to react when the browser window is resized, but document if we want to handle keypresses.

We want to start creating objects and calling their methods (ultimately to have the GPU render images for the canvas) when the webpage has loaded. This can be accomplished by registering an event listener function using window.addEventListener. We can pass a locally defined function that gets called one the webpage has loaded. We obtain the canvas and overlay objects, and create an object using the App class. This will be the only instance of App.

```
window.addEventListener('load', () => {
  const canvas = document.getElementById("canvas");
  const overlay = document.getElementById("overlay");
  overlay.innerHTML = "WebGL";

  const app = new App(canvas, overlay);
  app.registerEventHandlers();
});
```

App is responsible for creating a WebGL context associated to the canvas, and handling user and system events. Rendering content (and managing objects representing said content) are the responsibility of Scene — that is where we start to do actual graphics.

```
class App{
  constructor(canvas, overlay) {
    this.canvas = canvas;
    this.overlay = overlay;
    this.gl = canvas.getContext("webgl2");
    if (this.gl === null) {
      throw new Error("Browser does not support WebGL2");
    }
    this.scene = new Scene(this.gl);
    this.resize();
  }
}
```

The resize method is responsible for setting up the dimensions of the canvas. This functionality is separated because we need to do this one at the beginning, and every time the browser window is resized. clientWidth and clientHeight are the actual on-screen dimensions of the canvas, measured in pixels. The width and height properties define the *rendering resolution*, i.e. the size of the image computed by the GPU. If the physical and rendering resolutions do not match, the computed image must be stretched or compressed to fit the canvas. We avoid this by setting the rendering resolution properly. Calling scene.resize offers an opportunity to te scene object to perform tasks based on the dimensions of the canvas.

```
resize() {
  this.canvas.width = this.canvas.clientWidth;
  this.canvas.height = this.canvas.clientHeight;
  this.scene.resize(this.gl, this.canvas);
}
```

After the constructor, we call the registerEventHandlers method. This is separate from the constructor only for the purpose of organizing code a bit better. Here, we register an event handler to invoke the resize method when the window is resized. Note that the event handler is given as an arrow function, but we can omit the curly braces because the function body is a single expression. We also register a one-time event handler for rendering graphics content. Usually, we want the graphics content to change, meaning it must be redrawn periodically. The images appearing one after the other are like the frames of a film, displaying some kind of animation. That is why the method to register this handler is called requestAnimationFrame.

```
App.prototype.registerEventHandlers = function() {
  window.addEventListener('resize', () => this.resize() );
  window.requestAnimationFrame( () => this.update() );
};
```

Our update method is responsible for managing our graphics content (e.g. performing animation, simulation) and displaying it through WebGL calls. We should not call the update method directly, but let the browser decide when it is ready for drawing graphics. That is why we had to register our function that calls update when the animation frame must be drawn. We delegate actual rendering to the update method of the scene object, and trigger the next animation frame by registering an identical handler function.

```
update() {
  this.scene.update(this.gl);
  window.requestAnimationFrame( () => this.update() );
}
```

With this, our interfacing with the browser is completed, and we can proceed to implement Scene that manages objects of our virtual world and uses the WebGL API to render them. But before that, we must get acquainted with the GPU hardware.

Debugging

Now we are familiar with the browser running JS code for us. More often than not, however, the program we write just produces errors, or a behavior de did not intend. All modern browsers have an *inspection* or *debugging* mode, often available from the right-click context menu, that allows us to track these problems down.

The first tab we should check is the Console. We can enter JS code here which is immediately executed. Also, this is where log messages

and unhandled errors are printed. Where the error happened in the code is also displayed as a clickable link. Clicking it jumps to the Sources tab and opens the source file in question, highlighting the location or the error. Very often, the error message is going to be: Cannot read property <properyName> of undefined. This and similar error messages are a bit less informative as error messages displayed by compilers of strongly typed languages. This is because there is no checking for the type or number of function parameters. If those are wrong, the error does not happen where we call the function, but somewhere inside the function itself, trying to access properties of objects that are not the right type, or undefined. Therefore, it is important that we are able to follow the flow of operation through the chain of function calls that eventually lead to the error.

First of all, we need to pause the execution of the program just before the error happens. We can do this using a breakpoint. A breakpoint is placed on a line of code in the Sources tab by clicking by clicking the line number on the left. A blue label indicates the breakpoint, clicking which removes the breakpoint. If we reload the webpage, execution should stop at the breakpoint, highlighting the line. With script execution paused like that, we can investigate the state of all our variables. On the console, we can type in variable names or complete expressions, and see the result of their evaluation printed. We may even change values or add new properties, albeit these changes do not affect our actual code and are thus not permanent. There is a subwindow under the Sources tab with debugging tools. On the top, there are buttons to re-commence program execution, advance it by one line, advance it stepping into functions called, or step out of a function. These are very useful if we place the breakpoint not at a line causing the error, but somewhere earlier, and follow the operation of the program line by line, step by step. Under the buttons, we see Call Stack and Scope.

Scope displays all the variables that are in scope, i.e. local variables, including **this**, and global variables. Objects can be clicked to expand them and see all their properties. Thus, we are able to verify if the variables contain values which are consistent with the expected operation of our script. Variable values can also be seen and explored if we hover the mouse over an identifier in the source code, avoiding the need to look up the variable in the Scope list.

The call stack is the chain of function calls that took us to the current position in code, where execution is paused. As the bug is usually not in the function where the error happens, but in another function calling it, it is important that we are able to navigate this chain of function calls. Clicking a function moves the highlight to the line in that function that execution has reached. The scope is now

that of that function, and variables there can be inspected.

Thus, the process of fixing a bug is usually:

1. Place a breakpoint at or before the error.
2. Reload a the page.
3. Check values of variables by hovering the mouse over them, looking for a cause.
4. Check the values of the parameters passed to the function. If the problem appears to be with these, move up the call chain, and continue with verifying variable values there.
5. Once you have identified the cause, edit your source to fix the problem.
6. Remove the breakpoint.

Of course, debugging can be used in absence of explicit error messages, to investigate the workings of any suspect code. Usually, one proceeds along a sequence of expected operation, placing and removing breakpoints one after the other, narrowing down the location where the actual process deviates from the desired one. Often, whether a certain breakpoint is hit at all is an important indicator of proper operation.

Note that browsers may allow you to edit the source code in the Source tab. This has to be done carefully, and aware of some facts. Usually, edits in the browser only affect a temporary copy of the source, and are not reflected in the actual files. This can lead to confusing situations where we do not know which version is executing and how to revert to a consistent state. While browsers can be configured to commit source changes to the actual source files, the recommended practice is to avoid editing code in the browser altogether, and only use a dedicated text editor or development environment for coding or fixing bugs.

The GPU

The name *Graphics Processing Unit* has been used to market graphics cards, or on-board integrated hardware of the same purpose. It compliments the *Central Processing Unit* in that it also executes programs and performs computations. However, even if today's CPUs have several cores working in parallel, switching between multiple threads and processes, we are more or less right to think about the CPU as indeed a single unit executing one sequence of program operations at any one time. The GPU, on the other hand, includes several *multiprocessors*, with hundreds, or even thousands of *cores*, executing operations simultaneously. This high parallelism gives GPUs extraordinary processing power, but comes with severe limitations.

Dedicated GPUs have their own memory, called *device memory*, as opposed to *host memory*, which is accessible to the CPU. The cores are able to read from and write to this memory, but that is a slow process compared to the extensively cached memory accesses of the host memory by the CPU. GPUs can make up for this by some highly targeted special caching mechanisms (e.g. texture caching), but mainly by accessing memory in a *coalesced* manner. This means that cores on the same multiprocessor always simultaneously read and write data that is compactly aligned in memory. What is more, these cores share instruction control logic: they must execute the same operation (or leave some cores in the group idle, losing their performance edge). This architecture is only efficient if we have to process large buffers of data, element by element, independently, producing another contiguous buffer of output. Fortunately, this is exactly what we need to perform image synthesis using the *incremental* method, i.e. based on drawing lines and triangles.

The CPU works with the host memory, the GPU with the device memory.

GPUs are efficient when performing the same operation on array elements independently.

Is is possible to view the GPU as a collection of processor cores, running programs not unlike CPU cores, with the above limitations being mere guidelines to achieve high performance. That is what *General Purpose GPU*, or *GPGPU* frameworks like OpenCV or CUDA set out to do. However, GPUs offer specialized functionality for graphics, and an interface presented as a *pipeline* of various data processing *stages*. Some of those stages are in fact implemented

The GPU is a pipeline of processing stages.

by programs running on processing cores, but some are special circuitry for the specific purposes of image synthesis. Graphics libraries like Direct3D or OpenGL (including the embedded systems variant OpenGL ES, and its JavaScript implementation WebGL) operate with such a view of the GPU, and let us set up this pipeline to perform rendering tasks for us. We are allowed to run more limited programs on the processing cores (we call them *shaders*), with pre-determined input and output patterns, which ensures efficient parallelism without having to care about that explicitly.

The purpose of operating the GPU pipeline is rendering, i.e. creating an image. The image is represented as a finite resolution grid of rectangular cells, or *pixels*. Every pixel can have its own color. The image resides in GPU device memory, as a buffer of values specifying pixel colors. This area of memory is thus often referred to as the *color buffer*. The contents of the color buffer are typically presented on the screen of a display device to allow the user to see the image.

The pipeline output is an image in the color buffer.

The heart of the GPU pipeline is the *rasterizer*. This is specialized circuitry on the graphics card, ruthlessly efficient in one task only: rasterizing primitives. *Primitives* are, most often, triangles. Rasterization means drawing them by coloring rectangular cells (or *pixels*) of a finite resolution 2D grid. In fact, the rasterizer is only responsible for identifying the pixels to be colored, and not for actually picking their colors or writing those colors to anywhere in memory. Thus, it can be seen to break down triangles into pixel-sized *fragments*.

The rasterizer processes triangles.

The rasterizer generates fragments.

The functionality of all other stages of the pipeline can be seen as determined by the existence of the rasterizer. Considering the input side, it is triangles that we can drawn efficiently, so everything we need to draw must be somehow represented by triangles. Triangle corners are called *vertices*. The rasterizer needs to know the on-screen positions of the three vertices, so we need a stage to provide those: the *vertex shader*. Considering the output side of the rasterizer, the fragments produced need a color that can be written into pixels of the color buffer. The stage processing the fragments and producing those colors is the *fragment shader*.

The vertex and fragment shader stages are programmable. The programs they can execute are also called *shaders*. They are written in a shader language. Prominent shader languages are actually all very much alike, being based the C programming language, with GLSL (the OpenGL Shading Language) being the preferred choice for OpenGL, and HLSL (High Level Shading Language) for Direct3D. Much of the material of this course deals with what exactly the shaders have to do to compute the on-screen vertex positions or the pixel colors.

We already know that the output of the pipeline is a buffer of data,

specifically the color buffer. It is also often called the *target buffer*. The pipeline input is likewise stored in buffers residing in the device memory. Before the operation of the pipeline is started, we will need to fill these buffers with appropriate data. Actually, we may create several buffers to represent several different things we want to draw, and select the appropriate input buffer when drawing a specific *model*. Such a model may consist of some description of shape and color. As the rasterizer is only able to work with triangles, the shapes must be represented by *triangle mesh geometries*. The coordinates describing the positions of triangle corners (i.e. *vertex positions*) are the most essential data that should be given. As the data uploaded obviously describes vertices of the geometry, its storage in memory is called the *vertex buffer*. It is possible to list all vertex positions in the vertex buffer for all triangles, storing three sets of coordinates for every triangle, consecutively. However, as triangles representing a continuous surface often share vertices, this would waste memory, bandwidth, and computational power. If we list identical vertices only once in the vertex buffer, we need a different set of data to indicate which vertex triples span triangles. This array of integer indices is called the *index buffer*, where every triple of integers indicates three vertices in the vertex buffer. The GPU hardware is able to assemble the required input to the rasterizer using this data. Together, the vertex and index buffers define the *geometry*.

Input triangle mesh geometries are specified using vertex and index buffers.

A triangle mesh is often just the best approximation of the actual shape we want to draw. If we would like to render a sphere, but our polygon budget only allows for rendering twenty triangles, we will have to make do with a dodecahedron. But even if rendering just a dodecahedron, we could color the pixels covering its surface as if it was a sphere. This trick, known as *smooth shading*, can drastically improve image quality with low triangle counts. We discuss it in detail when we address shading of 3D surfaces, in chapter **??**. For smooth shading to be possible, we need extra information about the intended surface geometry, not just the vertex positions. Specifically, we need to store the *normal*, i.e. the vector perpendicular to the tangential plane touching the original smooth geometry, at every vertex. This allows us to render both a dodecahedron and a sphere, the vertex positions being identical, but the vertex normals different.

The vertex normals are thus also stored in vertex buffers, alongside the vertex positions. Both are called *vertex attributes*. While positions are practically always present, normals may or may not be required, depending on whether we implement smooth shading in our shader code or not. Thus, the attributes that describe a vertex are not set in stone, but they can be adjusted to our needs. This, as we might expect, requires additional settings telling the GPU how to

Shading normals may be vertex attributes.

interpret the data in our vertex buffers.

Positions and normals, along with the index buffer, define the shape of models. Their apparent color may also be given as a vertex attribute. This works great if there are relatively many vertices and the coloring does not need to have a lot of detail. However, low triangle count geometries can benefit greatly from *texturing*. Texturing means in-memory images can be applied on the triangle surfaces, not unlike decals. This, however, needs a per-vertex specification of what part of the decal image is supposed to end up on which triangle. That can be given in another vertex attribute, called *texture coordinates*. We discuss texturing in detail in chapters **??** and **??**.

Texture coordinates may be vertex attributes.

The same model can be drawn many times. For example, drawing one thousand spheres, at different locations or maybe even different radii, should not require creating and filling a thousand vertex buffers—they should all be drawn using the same model, with the same original vertex positions. Where these vertices end up on screen (i.e. where the rasterizer should draw them), changes when objects move; when they rotate; grow or shrink; but also if our entire view is panned or zoomed. Thus, calculations to this effect, i.e. considering object poses and view settings to find on-screen vertex positions, must be performed for each vertex. This is exactly what the vertex shader units can do. GPU processor cores run vertex shader code, processing all vertex data given in the vertex buffers, in parallel, independently. They write out results into a buffer in memory, from where the rasterizer is able to obtain, using the index buffer, the data for triangle vertices. We discuss how the computations should be performed by the vertex shader in chapters **??** and **??**.

The vertex shader is executed on all model vertices to find their on-screen positions.

Before fragments could be written into the pixels of the target buffer, their colors have to be found. This can be done in many ways, depending on our desired visuals. We write fragment shaders to formulate them. Often texturing and smooth shading techniques are involved in finding the fragment color. In any case, vertex attributes must be taken into account. Note that the rasterizer already needs to process the on-screen vertex positions of the triangle vertices, and produce the in-between positions for all the fragments. This process is known as linear interpolation. In fact, this is the essence of that the rasterizer does: churn out in-between positions for all fragments. This interpolation is extended to every output of the vertex shader. When writing our vertex shader code, we can choose to output any vertex attributes. These will get interpolated, and the intermediate values passed to the fragment shader as inputs.

In the above, we assumed that there is a mechanism to set some parameters and use them in the shaders: e.g. the view settings and object poses in the vertex shader, the texture in the fragment shader.

Note that these are different from the vertex attributes (i.e. the position, normal, or texture coordinates). Attributes have different values for all vertices. The above settings, on the other hand, are the same for all vertices. Therefore, they are called *uniforms*. Uniforms are set from the host program before the operation of the pipeline is started. Shaders can read, but not change them. Before drawing a second object, the host program may set different values for the uniform parameters, which then apply for all the processed vertices again.

Attributes are vertex shader parameters that are different, uniforms are the ones that are the same for all vertices.

Programming the GPU

In chapter , we went through the web programming requirements to produce an OpenGL context over a canvas in a web page, and request repeating opportunities to animate and move our virtual world. In this chapter, we continue by implementing Scene, which is responsible for these tasks. The Scene constructor is invoked by App when the web page has loaded, right after the OpenGL context has been created. This is where we should issue WebGL calls to create and initialize any GPU resources, including buffers in the device memory and shaders to be run by the GPU. It is possible to create many more of those than what can be used at any one time. Scene.prototype.update is called from App for every frame. This is both for animation and rendering. Thus, any changes to the virtual world should be applied, and WebGL calls should be issued to select a combination of some previously created resources and start the GPU pipeline operation to draw objects on the canvas.

We draw a single triangle first, in 2D, in solid color. Drawing more complex geometries are only a matter a supplying additional data. 3D graphics or more involved coloring requires writing adequate GLSL shaders and supporting them with parameters they require. There is an endless sea of possible rendering techniques, from the simple to the extremely complicated, that can be realized in shaders. This book covers the most important and widespread single-pass real-time techniques.

In order to render a triangle, we once need to

- create a *vertex buffer* with the three vertex positions,
- create an *input layout* (a.k.a. VAO, Vertex Array Object),
- create a *vertex shader* from GLSL source code,
- create a *fragment shader* from GLSL source code,
- and create a *program* linking the two shaders.

What they call a *program* in WebGL is the entirety of compiled shaders running on the GPU together. In our case, that is the ver-

tex and fragment shaders. Shaders must read and write data to specific hardware registers, and it is important that these input/output bindings match for the two shaders. Which input registers belong to which vertex shader inputs must also be part of the program.

In every frame, we can use the previously created resources to draw the triangle. We need to

- set the program to be used by the GPU,
- *draw* the geometry specified by the input layout (including which vertex buffers to use) and the index buffer.

We can realize the resource creating tasks in the Scene constructor, and the per-frame tasks in its update method. Instead of writing a wall of code in these functions, we create some helpers: TriangleGeometry for the buffers and input layout, Shader for shaders, and Program for programs. Their constructors are invoked for the Scene constructor to create the respective resources, and their methods are typically called from update to facilitate the use of said resources for drawing.

These helpers contain textbook examples of how to create a vertex buffer, index buffer, VAO, shader, or program. As WebGL functions do not operate independently, but their effects depend on the current *state*, which in turn depends on earlier function calls, the snippets of code in this chapter can be seen as atomic operations accomplishing a task.

The TriangleGeometry constructor must create the vertex buffer, and fill it with data.

```
1 this.vertexBuffer = gl.createBuffer();
2 gl.bindBuffer(gl.ARRAY_BUFFER, this.vertexBuffer);
3 gl.bufferData(gl.ARRAY_BUFFER,
4   new Float32Array( [
5     -0.5, -0.5, 0.5,
6     -0.5,  0.5, 0.5,
7      0.5,  0.0, 0.5, ] ),
8   gl.STATIC_DRAW);
```

Array buffer means vertex buffer in WebGL's less than fortunate nomenclature. The calls prefixed by `gl.` are all part of the WebGL API. First we create a buffer, and store ta reference to it in a new property called vertexBuffer. This name is arbitrary, but reflects our intent for this variable. Then, we select it into the WebGL state as the current array buffer (that is to say, vertex buffer). While JavaScript arrays are not typed, and numbers are not integers or floats but numbers, number representation matters if we have to interact with the hardware. Therefore, a Float32Array *typed array* is constructed from the regular JavaScript array. This can be passed as a parameter to bufferData, copying the numbers into device memory associated with this vertex buffer. Note that we just copied nine floats, without any indication of how they form vertices or how they should be used.

This vertex layout information must be specified in a *VAO*, Vertex Array Object. Once it has been created and set as current, we can explain the layout of *attributes* identified by integer IDs. Here we say that attribute 0 must be taken from **this**.vertexBuffer, 3 floats per vertex, with nothing else in the buffer. Here again, vertexAttribPointer does not have a parameter to specify which buffer to take the data from, but it understands it has to be the currently bound one.

```
  this.inputLayout = gl.createVertexArray();
  gl.bindVertexArray(this.inputLayout);

  gl.bindBuffer(gl.ARRAY_BUFFER, this.vertexBuffer);
  gl.enableVertexAttribArray(0);
  gl.vertexAttribPointer(0,
    3, gl.FLOAT, //< three pieces of float
    false, //< do not normalize (make unit length)
    0, //< tightly packed
    0 //< data starts at array start
  );

  gl.bindVertexArray(null);
```

Creating an index buffer is similar, but the WebGL name is *element array buffer*, and the buffer should contain integers. The interpretation of this data is straightforward: the three indices address the three vertices that span a triangle.

```
this.indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
  new Uint16Array( [0, 1, 2, ] ),
  gl.STATIC_DRAW);
```

With that, the geometry setup is complete. The Shader helper class' constructor perform standard steps of creating a WebGL shader.

```
this.glShader = gl.createShader(shaderType);
gl.shaderSource(this.glShader, Shader.source[sourceFileName]);
gl.compileShader(this.glShader);
if (!gl.getShaderParameter(this.glShader, gl.COMPILE_STATUS)){
  gl.getShaderInfoLog(this.glShader);
}
```

It creates a shader object, sets its source (a JavaScript string containing GLSL code), and compiles it to low level GPU operations. We can specify the source strings themselves in `.glsl` files.

Our initial vertex shader itself should just output the input position as the output position. It has an input of type **vec4** called vertexPosition. This input must come from the vertex attributes, but we have not yet said that this vertexPosition here is the same as the attribute 0, the layout of which we have recorded into the VAO.

```
#version 300 es
in vec4 vertexPosition;
```

```
void main(void) {
  gl_Position = vertexPosition;
}
```

gl_Position is a built-in compulsory output of the vertex shader. This is the position that the rasterizer unit will use.

Our initial fragment shader takes no inputs, but outputs a single **vec4** — the fragment color. The simplest is to return a fixed, solid color.

```
#version 300 es
  precision highp float;

  out vec4 fragmentColor;

  void main(void) {
    fragmentColor = vec4(1.0, 1.0, 0.0, 1.0);
  }
```

The Program helper links the shaders into a program. After creating the WebGL program instance, we can set the two shaders to it. It is here that we can bind the vertex shader inputs to concrete attribute numbers. With this, the data from the vertex buffer is associated with the vertex shader input variable.

```
this.glProgram = gl.createProgram();
gl.attachShader(this.glProgram, vertexShader.glShader);
gl.attachShader(this.glProgram, fragmentShader.glShader);

gl.bindAttribLocation(this.glProgram, 0, 'vertexPosition');

gl.linkProgram(this.glProgram);
```

Using the helper classes, the Scene constructor can elegantly create all resources.

```
this.vsIdle = new Shader(gl, gl.VERTEX_SHADER, "idle-vs.glsl");
this.fsSolid = new Shader(gl, gl.FRAGMENT_SHADER, "solid-fs.glsl");
this.solidProgram = new Program(gl, this.vsIdle, this.fsSolid);
this.triangleGeometry = new TriangleGeometry(gl);
```

Later, of course, we want to create more shaders, programs, and geometries—not to be used simultaneously, but one program with one geometry at a time. The helpers offer a way to do that without duplicating longish snippets of code.

Using the resources to draw our triangle should happen in update. That method is called whenever the virtual world needs to be redrawn, i.e. at repeating intervals.

We need to select our program as the currently used one:

```
this.gl.useProgram(this.solidProgram.glProgram);
```

We implement draw methods for JavaScript objects, where not only stored GPU settings should be applied, but also the operation of

the pipeline should be started. The first example is TriangleGeometry. Its draw method set its VAO, index buffer as current, and initiates drawing, giving the number of indices to process.

```
gl.bindVertexArray(this.inputLayout);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.indexBuffer);
gl.drawElements(gl.TRIANGLES, 3, gl.UNSIGNED_SHORT, 0);
```

With these, the update method can easily clear the canvas and draw the triangle.

```
gl.clearColor(0.6, 0.0, 0.3, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT);

this.solidProgram.commit();
this.triangleGeometry.draw();
```