Babylon.js Essentials

Are you familiar with HTML5? Do you want to build exciting games and web applications? Then explore the exciting world of game and web development with one of the best frameworks out there: Babylon.js.

Starting from the beginning, the book introduces the required basics for 3D development and the knowledge you need to use the Babylon.js framework. It focuses on the simplicity provided by Babylon.js and uses a combination of theory and practice. All the chapters are provided with example files ready to run; each example file provides the previously learned features of the framework. Finally, developers will be ready to easily understand new features added to the framework in the future.

Who this book is written for

Babylon.js Essentials is intended for developers who want to enter the world of 3D development for the Web, or developers who want to add the Babylon.js framework to their skill set. Familiarity with the notion of object-oriented programming would be helpful for understanding the architecture of the Babylon.js framework. Also, familiarity with web development would be useful, in order to understand the principles used.

What you will learn from this book

- Understand what the TypeScript language is and its benefits (compared to JavaScript) in large projects such as 3D engines
- Learn the basics of 3D using Babylon.js without too much theory but with an emphasis on practice, for a better understanding of the architecture
- Know the usage of Material—a fundamental principle of 3D engines in Babylon.js—and then customize the appearance of 3D objects
- Integrate collisions and physics in gameplay. Understand the notion of impostors for physics simulation
- Manage, create, and spatialize audio tracks in 3D scenes
- Go further with the Babylon.js framework to create actions on events
- Create rendering effects provided by the Babylon.js framework, such as post-processes
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Babylon.js and the TypeScript Language'
- A synopsis of the book’s content
- More information on Babylon.js Essentials
Julien Moreau-Mathis is a fan of 3D development. He started working with 3D development at the age of 17 and created a C++ framework named Community Play 3D. Now, he is a developer at Microsoft and he takes immense pleasure in being a part of the Babylon.js team.
Preface

3D development has always been something mystic. Starting from the beginning, this book introduces the required basics of 3D development and the knowledge you require to use the Babylon.js framework. It focuses on the simplicity provided by Babylon.js and uses a combination of theory and practice. All the chapters are provided with example files that are ready to run; each example file provides the previously learned features of the framework. Finally, the developers will be ready to easily understand the new features that are added to the framework in the future and use the more advanced features only using the documentation.

What this book covers

Chapter 1, *Babylon.js and the TypeScript Language*, provides a quick introduction to the Babylon.js story and a course on a fundamental that is the TypeScript language.

Chapter 2, *The Fundamentals of Babylon.js and Available Tools*, starts with the Babylon.js framework and creates the first 3D scene, showing the simplicity of the framework along with the theory.

Chapter 3, *Create, Load, and Draw 3D Objects on the Screen*, starts with the concepts of Chapter 2, *The Fundamentals of Babylon.js and Available Tools*, let's introduce the right way to create 3D scenes and work with 3D artists.

Chapter 4, *Using Materials to Customize 3D Objects Appearance*, explains the notion of materials in 3D engines. In other words, let's unleash the Babylon.js Standard Material.

Chapter 5, *Create Collisions on Objects*, focuses on the gameplay itself by managing the collisions in your scenes, including physics simulation.

Chapter 6, *Manage Audio in Babylon.js*, explains one of the added values of the Babylon.js framework. Let's add and manage sounds, including spatialized sounds, in your scenes.

Chapter 7, *Defining Actions on Objects*, introduces a smart way to trigger actions on the 3D objects themselves without to many lines of code as some tasks can be a pain for developers.

Chapter 8, *Add Rendering Effects Using Built-in Post-processes*, shows the preferred part of most of the 3D developers. This shows and explains how to easily beautify the 3D scenes using post-processes effects, combined with the simplicity provided by Babylon.js.
Chapter 9, *Create and Play Animations*, allows us to play with the animations system provided by Babylon.js. This chapter provides the final skill that you need in order to be ready to build your own professional 3D application!
Babylon.js is a framework that allows you to create complete 3D applications and 3D video games for the Web. Babylon.js has a community that grows day after day; a community that actively contributes to the project, adding more and more features. This chapter gives you a brief introduction to the framework's vision and the TypeScript language as, Babylon.js was developed using this.

The Babylon.js framework embeds all the necessary tools to handle specific 3D applications. It allows you to load and draw 3D objects, manage these 3D objects, create and manage special effects, play and manage spatialized sounds, create gameplays, and more. Babylon.js is an easy-to-use framework as you can set up (you'll see this later) these things with the minimum lines of code.

Babylon.js is a JavaScript framework developed using TypeScript. TypeScript is a compiled and multiplatform language that generates pure JavaScript files.

We will cover the following topics in this chapter:

- An introduction to Babylon.js
- The reason Babylon.js has been developed using TypeScript
- An introduction to TypeScript

**The creators**

Babylon.js was created by David Catuhe (@deltakosh), David Rousset (@davrous), Pierre Lagarde (@pierlag), and Michel Rousseau (@rousseau_michel). It's an open source
Babylon.js and the TypeScript Language

项目在他们的业余时间开发。当他们开始 Babylon.js 时，他们希望它设计为易于使用，并且能够为所有人提供一个可访问的 3D 引擎。官方网站（http://www.babylonjs.com/）包含了很多为初学者（甚至在3D）的教程，以及每个功能和场景的示例。

**Online tools provided by the Babylon.js solution**

Babylon.js 提供了几个在线工具来帮助开发人员和艺术家实验并尝试他们的作品：

- 对于开发人员，Playground（http://www.babylonjs-playground.com/）允许您进行实验和培训。它显示了一个带有自动完成的代码编辑器（Monaco）和画布，以查看结果。它还提供了一些示例代码供您培训。
- 对于艺术家，Sandbox（http://www.babylonjs.com/sandbox/）允许您拖动并拖放导出的 Babylon.js 场景（Blender 和 3ds Max）到浏览器以实时查看结果。Sandbox 提供了调试工具来激活/禁用功能并查看对实时性能的影响。
- Create Your Own Shader (CYOS) 允许开发人员开发着色器并在实时查看结果。也有几个着色器可供训练和实验。

**Why is Babylon.js developed using TypeScript?**

Babylon.js 是一个大项目，它从 GitHub 创建以来就不断增加贡献。它为您提供了很多功能，并且有时会有很多参数以获得更多的灵活性。TypeScript 语言非常有用，因为它旨在提高和确保 JavaScript 代码的生产。

**The TypeScript language**

TypeScript (TS) 是一个免费的开源语言，由微软开发。它是一种编译语言，可以生成 JavaScript（TS 代码是编译的）并提供一个静态类型系统，这是可选的。静态类型系统在 Babylon.js 中用于得到一个
cleaner and more descriptive code. It means that if a function has a lot of parameters, it's easier to fill and understand them instead of always using the documentation as a reference. Moreover, it allows developers to declare classes (as the ECMAScript 6 specifications do) and interfaces for a better understandable architecture and structure of code.

The TypeScript features

The typing system is powerful as it allows developers to create interfaces, enumerated types, and classes and handle generics and union typing. Overall, developers use the typing system for a better understanding and security of the libraries that they are building and using.

The TS language supports inheritance (classes) and also provides access specifiers (private / public / protected) to modify the access rights for the classes' members. Then, developers can see at a glance the members that they can use and modify.

Introduction to TypeScript – what you have to know

Let's introduce TypeScript with some feature examples and configurations: how to compile TS files to JS files, work with classes / types / union types, functions, inheritance, and interfaces.

Compilation using Gulp

Gulp is a task runner available as an npm package. It provides a plugin to handle the TypeScript compilation. The only thing to do is to configure a task using gulp with gulp-typescript.

To download the gulp packages, you have to install Node.js (https://nodejs.org/) to get access to the npm packages:

1. Install Gulp using the following command line:
   
   npm install gulp

2. Install Gulp-Typescript using the following command lines:
   
   npm install gulp-typescript
3. To configure the Gulp task, just provide a JS file named gulpfile.js containing the task description.

4. Import Gulp and Gulp-TypeScript:

   ```javascript
   var gulp = require("gulp");
   var ts = require("gulp-typescript");
   
   5. Define the default task to transcompile your TS files:

   ```javascript
   gulp.task('default', function() { // Default task
     var result = gulp.src([ // Sources
       "myScript1.ts",
       "myScript2.ts",
       // Other files here
     ]).pipe(ts({ // Trans-compile
       out: "outputFile.js" // Merge into one output file
     }));
     return result.js.pipe(gulp.dest("./")); // output file destination
   });
   
   6. Once the default task lists all the TS files to transcompile, just call Gulp using the following command line:

   ```
   gulp
   ```

Working with typed variables

Working with TypeScript is really similar to JS as the typing system is optional. Nevertheless, the common types in TS are as follows:

- String
- Number
- Boolean
- Any
- Void
- Enum
- Array

With JS, you should write the following:

```javascript
var myVar = 1.0; // or
```
```javascript
var myVar = "hello !";

Here, you can write exactly the same with TS. The TS compiler will process the type inference and guess the variable type for you:

```javascript
var myVar = 1.0; // Which is a number
// or
var myVar = "hello !"; // Which is a string
```

To specify the type of a variable with TS, type the following command:

```javascript
var myVar: type = value;
```

Then, with the previous example, add the following code:

```javascript
var myVar: number = 1.0;
// or
var myVar: string = "hello !";
// etc.
```

However, it's forbidden to assign a new value with a different type even if you don't mention the type as follows:

```javascript
var myVar = 1.0; // Now, myVar is a number
// and
myVar = "hello !"; // Forbidden, "hello" is a string and not a number
```

To get the JS flexibility with variables, let's introduce the `any` type. The `any` type allows developers to create variables without any static type. The following is an example:

```javascript
var myVar: any = 1.0; // Is a number but can be anything else
myVar = "Hello !"; // Allowed, myVar's type is "any"
```

The following is the screenshot of the `types.ts` file:

```
let this.trainWithTypes = () => {
  var notSpecified = 1.0;
  var specified: number = 1.0;
  var anySpecified: any = 1.0;

  notSpecified = "Hello !";
  anySpecified = "Hello !";
};
```

Let's introduce some specific types. It's the occasion to introduce the generics using
TypeScript and enumerated types. The usage of numbers, Booleans, and strings is the same in TypeScript and JavaScript. So, no need to learn more.

**Enumerated types**

Working with enumerated types (enum) is like working with numbers. The syntax is as follows:

```typescript
enum FileAccess {Read, Write};
```

This generates the following JS code:

```javascript
var FileAccess;
(function (FileAccess) {
    FileAccess[FileAccess['Read'] = 0] = "Read”;
    FileAccess[FileAccess['Writer'] = 1] = "Writer";
})(FileAccess || (FileAccess = {}));
```

Access to an enumerated type in both the languages is as follows:

```typescript
var myVar: FileAccess = FileAccess.Read; // Equivalent to 0
```

**Array**

Defining an array with TS is also similar to JS. The following is an example:

```typescript
// In both languages
var myArray = [];
// or
var myArray = new Array();
```

With TS, array is a generic class. Then, you can specify the item's type contained in the array as follows:

```typescript
var myArray = new Array<number>();
```

Note: With TS, typing `new Array()` is equivalent to `new Array<any>()`.

You can now access the common functions as follows:

```typescript
var myArray = new Array<any>();
myArray.push("Hello!");
```
myArray.push("1");
myArray.splice(0, 1);
console.log(myArray); // "[1]"

### Working with classes and interfaces

Classes and interfaces allow you to build types just as the `Array` class does. Once you create a class, you can create instances using the keyword `new`, which creates an object in the memory.

The following is an example:

```javascript
var myArray = new Array<any>(); // Creates a new instance
```

### Creating a class

The syntax in TS to define a class is as follows:

```typescript
class Writer {
    constructor() {
        // initialize some things here
    }
}
```

This generates the following in JS:

```javascript
var Writer = (function () {
    function Writer() {
    }
    return Writer;
})();
```

In both languages, you can create an instance of `Writer`:

```javascript
var myInstance = new Writer();
```

You can also use modules that work as namespaces:

```typescript
module MY_MODULE {
    class Writer {
    }
}
```

Access:
```javascript
var writer = new MY_MODULE.Writer(...);

Creating class members

With JS and the conventions, you can write the following:

```javascript
def function Writer() {
    this.myPublicMember = 0.0; // A public member
    this._myPrivateMember = 1.0; // A member used as private
}
```

With TS, you can explicitly specify the access specifier of a member (public, private, and protected), which has been explained as follows:

- **Public**: Any block of code can access the member to read and write
- **Private**: Only this can access this member to read and write
- **Protected**: External blocks of code cannot access the member; only this and specializers (inheritance) can access this member to read and write

Let's experiment using the Writer class:

```javascript
// declare class
class Writer {
    // Union types. Can be a "string" or an array of strings "Array<string>"
    public message: string|string[];
    private _privateMessage: string = "private message";
    protected _protectedMessage: string;

    // Constructor. Called by the "new" keyword
    constructor(message: string|string[]) {
        this.message = message;
        this._protectedMessage = "Protected message !"; // Allowed
    }

    // A public function accessible from everywhere.
    // Returns nothing. Then, its return type is "void".
    public write(): void {
        console.log(this.message); // Allowed
        console.log(this._privateMessage); // Allowed
        console.log(this._protectedMessage); // Allowed
    }
}
```

```javascript
var writer = new Writer("My Public Message !");
console.log(writer.message); // Allowed
```
Working with inheritance

Let's create a new class that specializes the `Writer` class. The specialized classes can access all the public and protected members of the base class thanks to the inheritance. The `extends` keyword represents the inheritance.

Let's create a new class named `BetterWriter` that specializes (extends) the `Writer` class:

```typescript
// The base class is "Writer"
class BetterWriter extends Writer {
    constructor(message: string|string[]) {
        // Call the base class' constructor
        super(message);
    }

    // We can override the "write" function
    public write(): void {
        if (typeof this.message === "string") {
            // Call the function "write" of the base class
            super.write();
        } else {
            for (var i=0; i < this.message.length; i++) {
                console.log(this.message[i]); // Allowed
                console.log(this._privateMessage); // Not allowed
                console.log(this._protectedMessage); // Allowed
            }
        }
    }
}
```

Using interfaces

Interfaces are used to create contracts. It means that if a class implements an interface, the class must provide all the functions and members defined in the interface. If not, it doesn't respect the contract, and the compiler will output an error.

All the defined functions are public and all the defined members are public.

With Babylon.js, a good example is to use the `IDisposable` interface. It means that the users can call the method named `dispose()`. This function's job is to deactivate and/or
deallocate the systems used.

The following is an example:

```typescript
interface IWriter {
    // The class "Writer" must have the "message" member
    message: string|string[];
    // The class "Writer" must provide the "resetMessages" function.
    resetMessages(): void;
}

class Writer implements IWriter {
    public message: string|string[];
    ...
    constructor(...) {
        ...
    }
    ...
    // All functions declared in the interface are public.
    public resetMessages(): void {
        this.message = this._privateMessage = this._protectedMessage = "";
    }
}
```

**Summary**

In this chapter, you obtained the necessary knowledge to develop programs using TypeScript with Babylon.js. You'll see that working with TypeScript can be more productive and secure in most cases. Additionally, some developers will be more comfortable when using types as they are used to development with typing.

Don't hesitate to manipulate TypeScript with the attached example files. Don't forget to install gulp and run the command lines.

You can also run the following command line:

```
gulp watch
```

This will track and recompile the TS files at each modification automatically.

In the next chapter, let's get straight to the heart of the matter with an introduction to the Babylon.js framework, and how to create an engine and scene entities such as lights, cameras, and meshes (3D objects). You'll build your first 3D scene with Babylon.js and understand the architecture of the framework really quickly!
Where to buy this book

You can buy Babylon.js Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.