Creating VR Worlds with Babylon.JS

By Brian G. Burton, Ed.D.

Published by Burtons Media Group, Abilene, Texas, United States of America
See http://www.BurtonsMediaGroup.com/books for more information.

Babylon.JS is an Open Source Project maintained by Microsoft©.

ISBN (eTextbook): 978-1-937336-20-2
Version 0.6.0 (8/02/2019)

About the Author

Brian Gene Burton, Ed.D. is a teacher, author, and game developer. Besides authoring books on mobile app development and contributing to several academic books on serious games and learning in virtual worlds, Dr. Burton has created game development degrees at two universities and enjoys researching and playing virtual environments. These programs are widely recognized with one of them ranked in the top 20% of game development programs.
Dr. Burton presents and publishes internationally on his research and enjoys sharing what he has learned about game and mobile development. When not traveling or teaching, he can be found at his home in the Ozark Mountains of Missouri with his beautiful wife of over 30 years, Rosemary. You can follow Dr. Burton's projects on http://www.BurtonsMediaGroup.com

Dedication

I dedicate this book to my loving wife, Rosemary, whose support and encouragement keeps us focused and writing. Thank you for keeping me focused and not running off on rabbit trails!

A special thank you to all of our students and the Babylon.JS community for their support and requests for specific details and editorial comments that helped so much with the development of this book.

Copy editing and formatting assistance provided by Brianna Rose Burton (http://www.LiteraryDiaries.com).

Preface

Welcome to the first book in the Virtual World Building series! When I began the discussion of creating a game development degree during the summer of 2004, I had no idea where it would take us or how much the industry would change in just than 15 years. The advancements in hardware and software have been astounding.
Now, with the advent of commercially viable Virtual Reality headset, we are on the cusp of a new era of innovation and opportunity. That is one of the reasons I became excited about the opportunities afforded to developers with Babylon.JS project. After reviewing the roadmap and chatting with some of the developers, I decided to dive deep into developing with Babylon.JS.

Why Babylon.JS?
For many years I have had a major project that has sat on the backburner. I had considered creating it with one of a number of game engines, but the slow pace of delivery and patching via app stores would supply the timely and rapid development needed to complete this project. That's where Babylon.JS comes in. It works. It's quick to deploy and follows industry standards. What more could I ask for?

While I am unable to cover everything that you might want to know in this first textbook on Babylon, hopefully you will find it useful to begin opening doors into further development for your pet project. If you have suggestions or would like to see something specific in a future release, please let me know!

Brian G. Burton, Ed. D.

Who this book is for

After a fair amount of discussion with colleagues and others trying to learn to develop for WebVR, we decided to make the assumption that the person using this textbook has very limited or no background in developing for web or games.
While this approach might frustrate the more experienced developer, hopefully the student who is learning web development, full stack, game development, or just learning to code for the first time, will find this approach helpful.

How This Book Is Organized

While writing this book, we have kept the traditional 15 or 16-week U.S. college semester in mind. Assuming one and a half chapters per week, this textbook should cover approximately half of a semester-long course. This textbook is not intended to cover everything that you should learn in a game development course. It is instead designed so that it would be used in an introductory course as part of the course requirements.

Conventions Used In This Book

Knowing that some eBook readers will automatically convert fonts to their system default, we have surrounded programming code with a box to distinguish it from the descriptive information in the book.

Using Code Examples and Fair Use Laws

This book was written to help you learn to develop applications and games with Babylon.JS. In general, you may use the code and examples in this book in your programs and documentation. You do not need to contact us for permission for reproducing a portion of the code. You don't need to ask permission to write an app that uses large chunks of code.
Now, on the other extreme, if apps appear that exactly reproduces the examples from this book, we will not be happy campers and will contact the app store that the offending app is a violation of copyright. We don't have issues with using the examples as a starting point, but take the app much further; be original! Answering questions by citing this book or quoting examples does not require permission (but we would appreciate the citation).
We reserve all rights for selling or distributing the examples in any format provided in this book. If you are not sure if your use falls outside of the fair use laws, please feel free to contact us at Sales@BurtonsMediaGroup.com.

Why didn't you use _____ for _____

There are a lot of great products available that can help the budding programmer/developer get their work done much faster. As this book is aimed at high school and college students, or people just getting started in game development, we tried not to use outside tools beyond those used in the basic Babylon Framework and referenced libraries. If you know of a great tool that can save time and money to developers, please share it with us and we might be able to include it in the next revision of the book.

How to Contact Us

Please address any comments or questions to DrBurton@BurtonsMediaGroup.com

How to succeed in web and game development

It has been widely stated that it takes 10,000 hours to become an expert at something. That works out to be five years spending an average of 40 hours a week, 50 weeks a year (we'll give you two weeks off each year for good behavior).

Game and web development are industries of passion. We are passionate about the websites and games we want to make. That passion must translate into time. Time spent working with the tools, understanding the process and creating games. Each of the tools covered in this book are tools that require a great deal of time to master. In a small studio, or as an Indie developer, you will usually have to have more than one area of mastery. It is our goal that this book will help you gain a basic understanding of each of the tools, enabling you to then pursue your passion. We recommend that you spend time every day with the software you will be using. Try to learn something new about the tool every time you sit down to work with it. Watch tutorials, read blogs and books (like this one) to help speed you along the path to mastery.

Outline/To the Teacher:

In creating this textbook, our goal was to create a step-by-step guide that introduces your students to the complex world of game development in manageable bites. You know your students' skills and what will motivate them better than we do, so please let us know how we can improve this resource to make your classes better. Below, we lay out the general outline and our reasoning for this methodology.
In **chapter 1**, the basic usage of Babylon.JS is introduced, including how to create a simple HTML file to load a scene.
Chapter 2 introduces the IDE, Babylon.JS Editor and how to create a new scene, save, and load scenes and projects.
Chapter 3

Creating this textbook has been like shooting at a moving target. The engineers at Babylon have done an amazing job and are constantly improving the framework. As there WILL be changes to the framework that impact this textbook, we will regularly update the content. Updates are **FREE** to anyone who has purchased a copy of the textbook through our website: www.BurtonsMediaGroup.com.

Pedagogy:

A note on the pedagogy used throughout this textbook. I am a constructivist by training and practice. I believe very strongly that students learn best by doing with appropriate scaffolding to enable them to be successful. For best results, **DO the examples** and the assignments. Reading the process used to accomplish a task is only a small part of the learning process. To fully realize and understand what you are trying to learn, doing the projects (or applying the concepts to your own project) is the best way to gain mastery of the knowledge being presented.

Resource Files:

All resource files (models, textures, and code samples) are downloadable from: https://github.com/drburton/babylonjs

Software:

All software referenced in this textbook is free and open source. Versions used for writing were:
Babylon.JS - Version 4.0
Babylon.JS - Version 2.5.3
Node.JS - Version 8.10

Table of Contents

Google Cloud
Microsoft Azure

Chapter 1: Basic Babylon.JS

For many (many) years, I have been developing applications and projects for the web. One of the projects that I had always wanted to create involved delivering learning content in a web browser that could be seen as a traditional 3D game environment, or, with the click of a button on the screen, become a fully immersive VR environment. Babylon.JS allows us to do just that!

What is Babylon.JS?

Babylon.JS is an open source JavaScript framework that can be used to build 3D experiences (games, training, data representation, learning activities, etc) that are played in your browser; desktop, mobile, or VR.
That's a long-winded way to say that with Babylon.JS, you can create experiences that are delivered by browser instead of being downloaded from an app store.

What You Will Learn

In this chapter, we will do a brief overview of Babylon, configure your computer so that you can test your projects on your local system, and complete our first project using JavaScript and HTML. Before we jump into building virtual worlds, let's review some key vocabulary that will be used in this chapter.

Key Vocabulary

- **HTML5** - the 5th version of HTML (HyperText Markup Language), which is used to define how a web page is loaded in a browser.
- **JavaScript** - a high-level interpreted programming language, widely used in the development of internet applications.
- **JavaScript Library** - a collection of pre-written JavaScript code that provides functions that can be called and used in development.
- **JavaScript Framework** - provides a design to develop an application.
- **TypeScript** - an open source high-level interpreted programming language that is a superset of JavaScript. TypeScript can be transcompiled into JavaScript. Used for large applications.
- **Scene** - the virtual environment that you are building. There can be multiple scenes to a virtual environment
- **Scene Graph** - The data structure of the scene; the object comprising a scene.
- **WebGL** - Web Graphics Library - a JavaScript library for rendering 2D & 3D images in a browser. Babylon.JS simplifies WebGL, making it easier to build virtual environments.

- **WebVR/WebXR** - Web Virtual Reality/Web Augmented/Mixed/Virtual Reality - A JavaScript library for rendering virtual, augmented, or mixed reality. Requires the appropriate equipment such as Oculus Rift, Oculus Go, HTC Vive, Windows Mixed Reality, or a mobile device. WebXR replaced WebVR as the standard in late 2018.

Why Babylon.JS?

After reviewing the vocabulary, you might have the question, why Babylon.JS? Why don't we just use WebGL and WebVR/WebXR?

Babylon.JS provides a JavaScript framework enabling the rapid development of 3D environments using WebGL/WebXR libraries. While it is possible to build your own framework using WebGl/WebVR libraries, you will spend most of your time creating the framework instead of creating the 3D world of your dreams. In some situations, that is appropriate, but not for what we will be doing in this class.

While reviewing different WebVR capable frameworks, I quickly became impressed with Babylon.JS over other frameworks because… well, it just works. Other frameworks had difficulty importing models, handling cloth, consistent frame rates, or PBR.

With Babylon.JS, all of those features worked and worked well with little effort on the part of the developer.

Adding in the open source nature of Babylon.JS and that the development is backed by Microsoft, it was an easy choice to make once all the points were tallied.

Online Resources

There are many excellent resources available for Babylon.JS. Here are a few to help you get started, or where to go should you become stuck:

BabylonJS.com - is the official website for Babylon.JS. A great starting point for becoming more familiar with the framework.
Babylon-playground.com - provides a site to experiment and see examples of Babylon.JS in action.
Sandbox.babylonjs.com - is a quick way to test 3D models to ensure that they will properly import into Babylon.
Editor.babylon.js.com - browser-based editor for Babylon.JS (discussed in Chapter 2).
github.com/BabylonJS/Babylon.js - GitHub repository of the latest version of Babylon.JS.
Forum.babylonjs.com - The official forums for Babylon.JS.
github.com/drburton/babylonjs - GitHub repository for code samples from this textbook.

Installing Babylon.JS

Installation of Babylon.JS on your local computer is not required unless you plan to work offline. You can download the latest version of Babylon.JS from the GitHub repository listed above. All examples were completed using Babylon.JS v. 4.0.

While it is not a requirement to download and install Babylon.JS to your local computer, there are some great resources such as skyboxes, ground textures, and other useful tools included in the GitHub repository.

Installing a Local Web Server

To speed up development, it is useful to have a local web server configured to allow you to view your development progress without having to upload everything to a remote server. When working with the editor, the recommended toolset to use is Node.JS. We have provided the process of installing both types of Node.JS web servers in the Appendix.

Of course, you can also just enter the file URL in the browser window, too:
file://c:/folder/myhtmlfile.html

Scripting with TypeScript

With Babylon.JS, the preferred scripting language is TypeScript, though you can also complete code in JavaScript.

TypeScript is a superset of JavaScript, providing greater control and security for your applications. To use a TypeScript set of instructions, it must be transcompiled into JavaScript to be used in your project. While the first few examples will be done in JavaScript, the majority of this textbook will be focused on using TypeScript. Don't worry if you are not familiar with JavaScript or TypeScript, we will cover how to use both.

JavaScript and TypeScript both use a semicolon (**;**) to designate the end of a command line, so you will sometimes see a long command spread over several lines on the screen.

Script Editor

Script creation is best done in an editor. While there are many great editors out there, I am going to focus on using Visual Studio Code: https://code.visualstudio.com/. Be sure to select support for JavaScript and TypeScript when you first start Visual Studio to make editing and transcompiling easy.

Your First Project

The best way to learn is by doing. So, we will begin by creating a simple scene using Babylon.JS. We will go into the anatomy of a complex project in a later chapter. For now, let's examine the components of a simple Babylon.JS project. We will be using JavaScript in this example rather than jumping into TypeScript.

This project is composed of three (3) parts: HTML, Babylon.JS Framework, and the Scene.

Basic Index.html

Below is a simple HTML script that sets the page title, defines the style (using 100% of the browser window) and a render canvas for Babylon to display the project.

```
<!DOCTYPE html>
<html>
   <head>

      <title>Hello World using Babylon js</title>

      <!-- Babylon.js -->
            <!-- Babylon.JS Framework Goes Here -->

      <style>
         html, body { width: 100%; height: 100%;  }

         #renderCanvas {  width: 100%; height: 100%; touch-action: none; }
      </style>
   </head>
<body>
   <canvas id="renderCanvas"></canvas>
   <script>
        //Babylon.JS Scene script goes here.

   </script>
</body>
</html>
```

Note that I have included bolded comments in two (2) places to show where the required scripts to load Babylon.JS and the scene scripts will be added.

Adding Babylon.JS Framework

To load the Babylon.JS framework replace **<!-- Babylon.JS Framework Goes Here -->** with the following command:

```
<script src="https://cdn.babylonjs.com/babylon.js"></script>
```

This will cause the server to load Babylon.JS. In the future, we will add additional libraries and frameworks to make our development easier. For now, all we need is the one script.

In the second location of our HTML file, replace **//Babylon.JS Scene script goes here.** With the following JavaScript.

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);

var createScene = function() {
    // Create scene
    var scene = new BABYLON.Scene(engine);

    return scene;
};

var newScene = createScene();

engine.runRenderLoop(function () {
    if (newScene) {
        newScene.render();
    }
});

// Resize
window.addEventListener("resize", function () {
    engine.resize();
});
```

Let's examine what this script is doing:

var canvas = document.getElementById("renderCanvas");

Using the **var** (short for variable), JavaScript creates a variable named **canvas**. **document.getElementById** looks at the HTML above and finds **renderCanvas**. This has the effect of telling Babylon.JS to setup the a display area (the canvas) using the renderCanvas parameters provided in the HTML.

var engine = new BABYLON.Engine(canvas, true);

Once again, we are creating a new variable, this one is called **engine**. **Engine** is assigned the value of the Babylon.JS engine, which includes a bunch of important operations that we will explore later, but for now, know that the engine will be working with the canvas (set in the HTML above), and the **true** option tells the engine to adapt to the device it is being displayed on.

var createScene = function() {

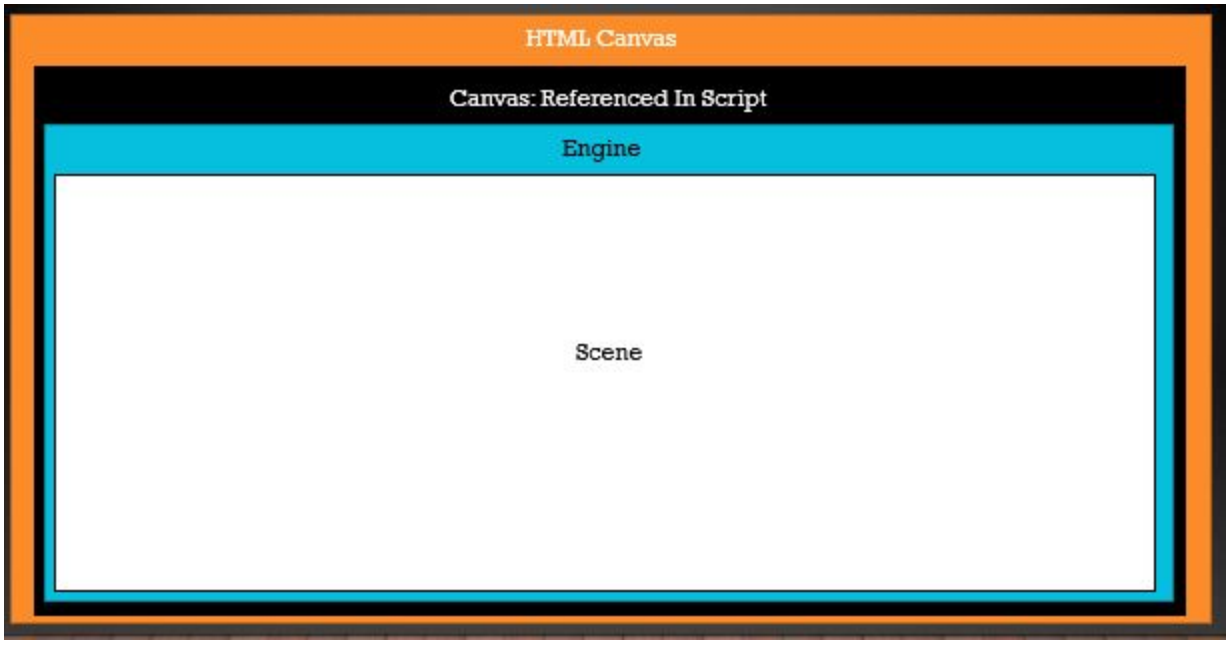
```
        // Create scene
        var scene = new BABYLON.Scene(engine);

        return scene;
    };
```

The next command is best seen as a group. We start by creating a new variable called **createScene**. CreateScene is going to be a function.

A **function** is a set routine that can be used multiple times. It will only be used when it is called by name i.e. the commands that are stored inside the function (any commands within the curly brackets **{ }**).
Within the function, we have a comment. Comments in JavaScript or TypeScript start with **//** - commenting everything to the right.
Next, a new variable **scene** is created to store the scene (i.e. what is showing on the screen - everything that the user might see or interact with).

With the program code, we have created a chain of commands that are interdependent:



First, the HTML canvas is defined in the HTML portion of our script.
The Engine draws upon the Canvas to define the working area for the virtual environment.
Then, the Scene uses the Engine to pass everything that is stored in the scene environment (which is currently nothing, but we will rectify that in just a few minutes).

The last command of the function **return scene;** does just that, returns the information stored in the scene back to the Babylon.JS engine so that it can be rendered onto the canvas.

var newScene = createScene();
Remember that we mentioned functions have to be called before they can do anything? Time to call the function! The **return scene** in the function will send back everything stored in the

scene to the calling function. We are going to store that as **newScene**. Anytime you need to call a function, just enter its name, in this case, **createScene();**
The parenthesis can be used to pass information to the function if needed (we don't need to this time, but we still need the parenthesis).

Now, it is time to start our rendering loop. All games and virtual environments are dependent on a rendering loop to continuously draw the scene until it is told to stop. How quickly it draws the environment is what gives us the frames per second (fps) rating of a game engine. Babylon.JS is optimized for 60fps. Depending on the quality of the computer, it does a pretty good job of handling large scenes and maintaining that goal.

```
engine.runRenderLoop( function () {
        if (newScene) {
            newScene.render();
        }
    });
```

The **engine.runRenderLoop** creates a new function. This function looks to see if newScene exists. If it does, then it renders the scene. There are a variety of ways to handle a rendering loop, but this will handle everything that we need to do for now.

If is used to check to see if a situation evaluates to true or false. The simplest version of if (such as the one we have above) will run any commands in the curly brackets **{ }** after the evaluation. So how can **newScene** be **true**? In programming languages such as JavaScript, we can check to see if a variable has value, or exists. If it does, then it is true… i.e., it has value. If it did not exist (there was not a previous var newScene command), or did not have a value assigned to the variable, then it would be false.

```
// Resize
window.addEventListener("resize", function () {
        engine.resize();
});
```

The last set of commands is a nice little piece of script to handle the user changing the size of the browser window. It uses what is called an event listener to check for a resize event. If the player/user does resize the window, then the engine will be told to resize the scene so that it still fits on the canvas.

Basic Shapes

Now that we have the basic framework of a Babylon.JS script, we need to add a camera, lights, and something to render to:

New JavaScript code is **bolded**:

```
var createScene = function() {
    // Create scene
    var scene = new BABYLON.Scene(engine);
```

```
        // Add a camera to the scene and attach it to the canvas
        var camera = new BABYLON.ArcRotateCamera("Camera", Math.PI / 2, Math.PI /
2, 2, new BABYLON.Vector3(0,0,5), scene);
        camera.attachControl(canvas, true);

        // Add lights to the scene
        var light1 = new BABYLON.HemisphericLight("light1", new BABYLON.Vector3(1,
1, 0), scene);

        return scene;
    };
```

Let's review each of these in order:

Cameras:

A camera is critical for the engine. Without a camera, you do not have a view into the virtual environment. It would be like trying to watch a movie, but there were no cameras used. You get a blank screen!

There are several types of cameras available in Babylon.JS.

- **ArcRotateCamera** - faces the target and orbits around the target location or object.
- **UniversalCamera** - traditional camera used for a first or third person view.
- **FollowCamera** - follows a target mesh as it moves through the environment. Can also be used as **ArcFollowCamera**, combining the capabilities of an ArcRotateCamera that will follow a targeted mesh.
- **AnaGlyphCameras** - for use with red and cyan 3D glasses.
- **DeviceOrientationCamera** - the camera responds to a mobile device orientation feedback.
- **VirtualJoystickCamera** - displays on-screen joysticks to control the camera view.
- **VRDeviceOrienationCamera** - Works with VR device feedback to change the orientation of the camera.

In this example we are using the **ArcRotateCamera** to rotate around a point in space or a mesh.

var camera = new BABYLON.ArcRotateCamera("Camera", Math.PI / 2, Math.PI / 2, 2, new BABYLON.Vector3(0,0,5), scene);

The parameters that make up the ArcRotateCamera are:

Camera: Name of Camera in Scene Graph,

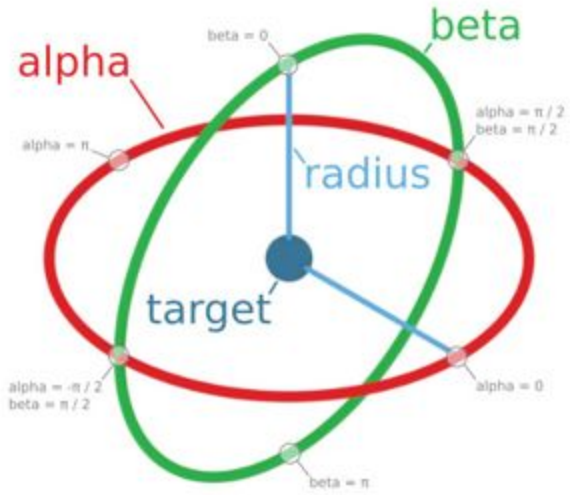
Math.PI/2: alpha rotation of the camera on the longitudinal (horizontal) axis,

Math.PI/2: beta rotation of the camera along the latitudinal (vertical) axis,

2: radius distance from target,

Babylon.Vector3(0, 0, 5): target,

scene: scene



Rotation for Arc Rotate Camera Source: BabylonJS.com

What is **Vector3**?
A Vector3 is used in 3D geometry to provide a location or coordinate in space. Usually supplied as the X, Y, and Z location. The value can be given as a whole number or decimal. When an object is created, the center of the object is set at the world axes (0, 0, 0) unless it is given a different location at the time of creation. If this doesn't make a lot of sense, don't worry, we will talk about it more in the next chapter.

Once the camera is added, it needs to be attached to the controls (i.e., the mouse and keyboard):
camera.attachControl(canvas, true);
The attachControl method connects the camera to receive input from the canvas, the "true" parameter tells the camera to ignore any keystrokes or mouse movements that are not defined.

Now that we have a camera, we need to light up the scene.

Lights:

Just as we have a variety of cameras for different needs, there are also several types of lights. There are four (4) types of lights:

- **Point Light** - light is emitted from a point in the scene space, like a light bulb.
- **Directional Light** - emits the light for everywhere toward a specific direction. You can think of it as the sunlight from a sunrise, or sunset.
- **Spot Light** - like a flashlight or spotlight, the light is from a specific point in one direction.
- **Hemispheric Light** - full ambient lighting of the scene.

Lights can also have color properties applied, allowing for many possibilities. You can also control the light's intensity with the intensity property (setting a value of 0 to 1).
For our application, we are using a hemispheric light
var light1 = new BABYLON.HemisphericLight("light1", new BABYLON.Vector3(1, 1, 0), scene);

The parameters for the hemispheric light are:

- **"light1"** - Name of the light in the scene graph
- **new BABYLON.Vector3(1, 1, 0)** - Direction of the light
- **scene** - Scene to place the light

Meshes

Let's add something for the camera and lights to show. Add the **bold** script to your file:

```
    var createScene = function() {
        // Create scene
        var scene = new BABYLON.Scene(engine);

        // Add a camera to the scene and attach it to the canvas
        var camera = new BABYLON.ArcRotateCamera("Camera", Math.PI / 2, Math.PI / 2, 2,
new BABYLON.Vector3(0,0,5), scene);
        camera.attachControl(canvas, true);

        // Add lights to the scene
        var light1 = new BABYLON.HemisphericLight("light1", new BABYLON.Vector3(1, 1,
0), scene);

       // Add some shapes to look at
        var sphere = BABYLON.MeshBuilder.CreateSphere("sphere", {diameter:2},
scene);
        sphere.position.y = 2;

        var cube = BABYLON.MeshBuilder.CreateBox("cube",{size:1},scene);
        cube.rotation.x = 2;
        cube.rotation.y = 3;
        cube.rotation.z = 4;

        return scene;
    };
```

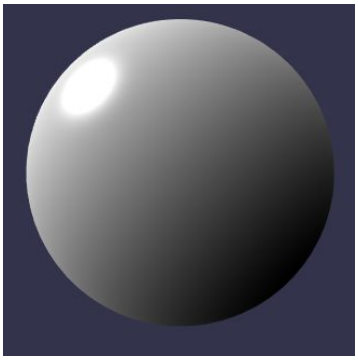
BABYLON.MeshBuilder has a variety of meshes that can be loaded with a simple command. Here we have loaded two (2): a sphere and a box.

First the sphere:
var sphere = BABYLON.MeshBuilder.CreateSphere("sphere", {diameter:2}, scene);
The sphere is a simple mesh, creating a sphere for the scene environment.
You just need to provide a name for the scene graph ("sphere" in this case), the size of the

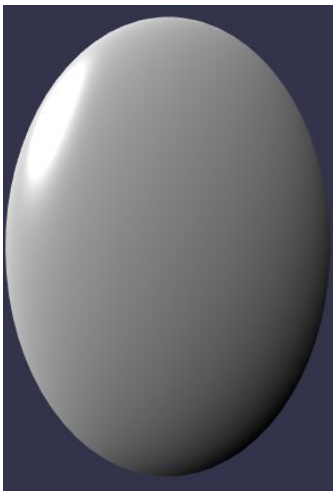
diameter, and the scene name to load the sphere into. Note that the diameter is surrounded by curly brackets { }. The brackets are used to include additional possible parameters. Other optional parameters include:

- segments - number of horizontal segments (default is 32)
- diameter - diameter of the sphere (default is 1)
- diameterX - diameter of the X-axis (allows for creating oblong spheres)
- diameterY - diameter of the Y-axis (allows for creating oblong spheres)
- diameterZ - diameter of the Z-axis (allows for creating oblong spheres)
- arc - ratio of the circumference between 0 and 1 (default is 1)
- slice - ratio fo the height between 0 and 1 (default is 1)
- updatable - can the mesh be updated? (default is false)
- sideOrientation - which side is oriented toward the scene

To include additional parameters, just add them to the curly brackets. So instead of
{diameter:2}



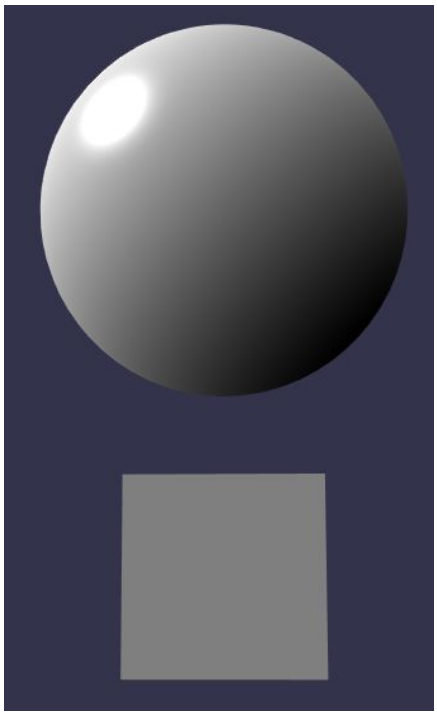
You could have:
{ diameterX:2, diameterY:3, diameterZ:1 }
Giving an oblong shape to our sphere:



After we load the sphere, we move it a little higher in the scene to make room for the box by adding the command
sphere.position.y = 2;

All meshes can be adjusted in a scene by changing their x, y, or z position.

The box is very similar to the sphere:
var cube = BABYLON.MeshBuilder.CreateBox("cube",{size:1},scene);

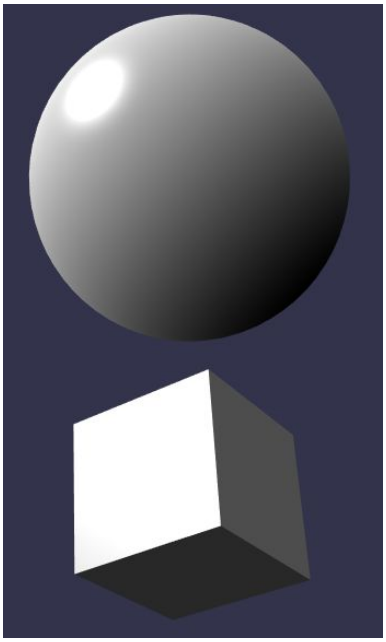


Possible parameters for a box include:

- size - box side size (default: 1)
- height - height of box (default is the value of size)
- width - width of box (default is the value of size)
- depth - depth of box (default is the value of size)
- faceColors - array of 6 Color4 (default: Color4(1,1,1,1) for each side - i.e. white with an alpha of 1).
- faceUV - array of 6 Vector4 (default: UVs(0,0,1,1)), used to apply texture or color to individual faces of the box mesh.
- updateable - is the mesh updatable? (default: false).
- sideOrientation - number of the side oriented forward (default: DEFAULTSIDE).

What is **Color4**? Color4 holds a color using Red, Green, Blue, Alpha (RGBA) to represent the color. Color4(R, G, B, A). Each is represented by a value between 0 and 1.

Now that we have the box on the screen, let's add a little rotation so that we can see more of the faces when it first starts. Rotation is by default 0. Changes are made using the radian system instead of degrees (1 rad = 57.3 degrees).
To apply rotation, supply a value (in radians) to the rotation property of the mesh:
cube.rotation.x = 2;
cube.rotation.y = 3;
cube.rotation.z = 4;



Other common meshes that can be loaded using MeshBuilder include:

- Cylinder - **CreateCylinder**("*name*", {*options*}, scene) - creates a cylinder mesh
- Decal - **CreateDecal**("*name*", sourceMesh: *MeshName*, {*options*}); - creates a decal mesh that can be applied to a model
- Disk - **CreateDisc**("*name*", {*options*}, scene); creates a disc shaped polygonal mesh
- Ground - **CreateGround**("*name*", {*options*}, scene); - used to create a ground mesh
- Ground from a height map - **CreateGroundFromHeightMap**("*name*", url, {*options*}, scene); - used to create a height map for the ground mesh using an image (the url).
- Ico Sphere - **CreateIcoSphere**("*name*", {*options*}, scene); - used to create a 20 sided object, (i.e. a 20 sided dice). Can also be used to make additional complex shapes.
- Plane - **CreatePlane**("*name*", {*options*}, scene); - used to create 2D plane
- Tiled Ground - **CreateTiledGround**("*name*", {*options*}, scene); - used to create a tiled ground mesh

You can see additional meshes and the options available for these meshes at https://doc.babylonjs.com/api/classes/babylon.meshbuilder

Now that we have our meshes loaded, we can shift gears and give them some color and texture.

Materials

And now to add some color and texture. Color and texture are two different things, but both are referred to as a material and applied to a mesh or object to change the appearance of the object.

As before, add the **bold** script:

```
        var createScene = function() {
            // Create scene
            var scene = new BABYLON.Scene(engine);

            // Add a camera to the scene and attach it to the canvas
            var camera = new BABYLON.ArcRotateCamera("Camera", Math.PI / 2, Math.PI / 2, 2,
new BABYLON.Vector3(0,0,5), scene);
            camera.attachControl(canvas, true);

            // Add lights to the scene
            var light1 = new BABYLON.HemisphericLight("light1", new BABYLON.Vector3(1, 1,
0), scene);

           // Add some shapes to look at
            var sphere = BABYLON.MeshBuilder.CreateSphere("sphere", {diameter:2}, scene);
            sphere.position.y = 2;

            var cube = BABYLON.MeshBuilder.CreateBox("cube",{size:1},scene);
            cube.rotation.x = 2;
            cube.rotation.y = 3;
            cube.rotation.z = 4;

            var materialSphere = new BABYLON.StandardMaterial("texture1", scene);
            materialSphere.diffuseColor = new Babylon.Color3(1,0,0);
            materialSphere.alpha = 0.3;

            sphere.material = materialSphere;

            var materialCube = new BABYLON.StandardMaterial(“texture2”, scene);
            materialCube.diffuseTexture = new BABYLON.Texture(“ground.jpg”, scene);

            cube.material = materialCube;

            return scene;
        };
```

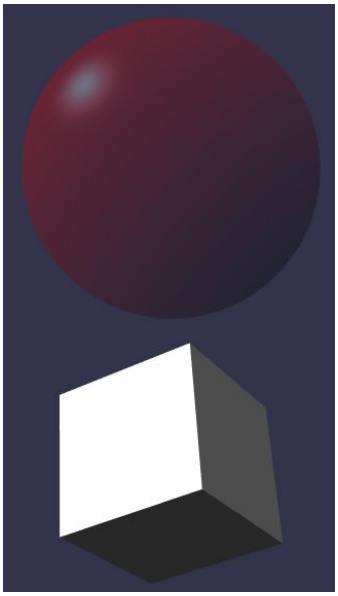
Adding color to a mesh
The first example will add color to the mesh. To get started, we will need to create a variable and tell Babylon that it will be used to store color information for a mesh:
var materialSphere = new BABYLON.StandardMaterial("texture1", scene);

As with previous examples, we are telling the system that the variable materialSphere will store standard material information. It will be named “texture1” in the scene graph, and it should be associated with the scene.

Once the variable is created, we can then change the **diffuseColor** property of our materialSphere variable and set it equal to a Color3 color. A Color3 passes three (3) parameters, representing R(ed), G(reen), B(lue). In this case, we are passing the color red.
materialSphere.diffuseColor = new Babylon.Color3(1,0,0);

There are many additional properties available to us. One is the **alpha** property, which can be used to adjust the transparency. The closer to zero, the more transparent the color will be.
materialSphere.alpha = 0.3;

Once we are happy with the settings, we can apply the material to the sphere by setting the sphere’s material property equal to the materialSphere variable:
sphere.material = materialSphere;



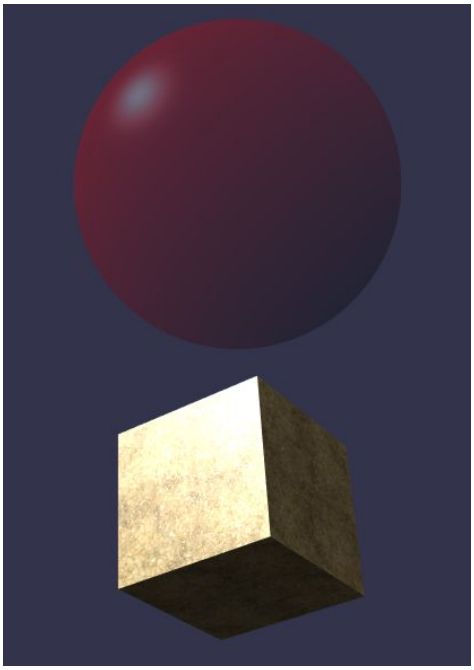
Adding a texture to a mesh
Now let’s add a texture. A texture uses an image file (png, jpg, or ktx) to create the appearance that the object has texture (wood grain, concrete, steel, etc). The file can be loaded from the project work area or a URL. As we did with the color, we first create a variable to hold the texture.
var materialCube = new BABYLON.StandardMaterial(“texture2”, scene);

Then load the file into the diffuseTexture property:
materialCube.diffuseTexture = new BABYLON.Texture("ground.jpg", scene);

A **diffuse texture** is the main color or image as it would appear under white light. You can also include:

- **Specular color** (specularColor) - Shows where the light is most intense
- **Emissive color** (emissiveColor) - Light (color) coming from the object.
- **Ambient color** (ambientColor) - The color picked up from the scene (ie., light reflecting off of a red object will give a red ambient reflection). Note, the ambient color must also be set for the scene.

Once you have your texture loaded, you are ready to set it to the mesh:
cube.material = materialCube;



Note: If you are loading the file in the browser instead of using a local web server, the texture will not apply properly to the cube due to safety limitations built into the browser.

In a later chapter, we will explore the use of different types of textures such as PBR. For now, enjoy your new scene!

WebVR

Yes, Babylon supports Virtual Reality in your Browser! Just add one line of code to the project before the **return scene;** command:

```
        // Enable VR
        var vrHelper = scene.createDefaultVRExperience({
createDeviceOrientationCamera: false });
```

This one line enables the VR option for your site. In a later chapter, we will add the ability to teleport around the environment.

Show the World

When you are happy with your project, it's time to show the world what you have been creating! Publishing to a remote server isn't difficult. In the Appendix, the process to create a website using various cloud platforms is discussed.

Once you have a web hosting service configured, you will need to upload the files for your project to the web host using an FTP client. There are a variety of programs available, so pick one that you are comfortable using.

When uploading, you will need to upload the HTML file and any image files (such as "ground.jpg") used in the HTML file. By default, if a browser is not given a specific web page as a part of the URL, then it will attempt to load index.html.

I named the file used in the project "CVRW-Ch1-HelloWorld.html". If I do not change the name, then to load the page, I would have to enter the full URL:
www.coolwebsite.me/CVRW-Ch1-HelloWorld.html
to access the webpage.

Summary

In this chapter, we introduced the basic framework for creating a virtual environment within a browser. That included how to load the Babylon.JS framework, setting up the canvas, engine, and scene, then loading a camera, lights, and a few basic mesh objects. In the next chapter, we will shift to using the Babylon.JS editor, which will handle some of the heavy lifting for us.
Questions

Define the following terms:
Color3
Scene
Vector3
Mesh
ArcRotateCamera

1. What does it mean to **transcompile**?

2. Research the differences between TypeScript and JavaScript?
3. What is the difference between hosting a web page locally vs on a remote server?
4. Explain the difference between a diffuse color and a diffuse texture.
5. Explain a Scene in Babylon.JS.

Assignments

1. Load a different shape such as a ico-sphere or a cylinder into the scene.
2. Change the sphere mesh to blue.
3. Add an additional box and place it beside the first box. Make it green in color.
4. Add a second sphere to the scene and place it above the first sphere.
5. Change the texture of the box to a different jpg or png image.
6. Create an IcoSphere in the scene: BABYLON.MeshBuilderCreateIcoSphere(“ico”, {subdivisions: 1}, scene); Adjust the subdivisions parameter and place it to the side of the box.

Chapter 2. Using the Editor

What You Will Learn

In the first chapter, we covered how to create a simple scene using JavaScript with the Babylon.JS Framework. In this chapter, we are going to begin using the Babylon.JS scene editor to create more complex scenes.

The Editor

Before you jump into the Babylon.JS editor, you might need to adjust your thinking. The goal of the editor is to simplify interaction with Babylon.JS and interaction with other common JavaScript Libraries such as Cannon.JS and Oimo.JS.

If you have been developing in a game engine, you will find that the editor is NOT a game engine. Babylon.JS editor is a Scene Editor. The editor allows you to code, work with materials and textures, apply physics, move objects, apply lighting, apply animations, and many other time-saving features.

Two Versions of the Scene Editor

There are two versions of the Babylon.JS editor: 1) a browser-based editor (http://editor.babylonjs.com/) and a desktop version (via electron.js).

- Windows: http://editor.babylonjs.com/BabylonJS Editor.exe
- Mac OS X: http://editor.babylonjs.com/BabylonJS Editor.dmg
- Linux: http://editor.babylonjs.com/BabylonJS Editor.zip

The Babylon.JS editor is a community project, and not maintained by the developers at Microsoft. A big thank you to a great bunch of people! Development on the editor continues at a rapid pace. We have found that the team working on the project (led by Julien Moreau-Mathis) easy to work with and dedicated to making the editor the best possible tool for Babylon.JS.

The online editor has the limitation that it must save to a cloud-based resource (Microsoft OneDrive). To have full access to your local file system, we recommend that you use one of the desktop-based editors.

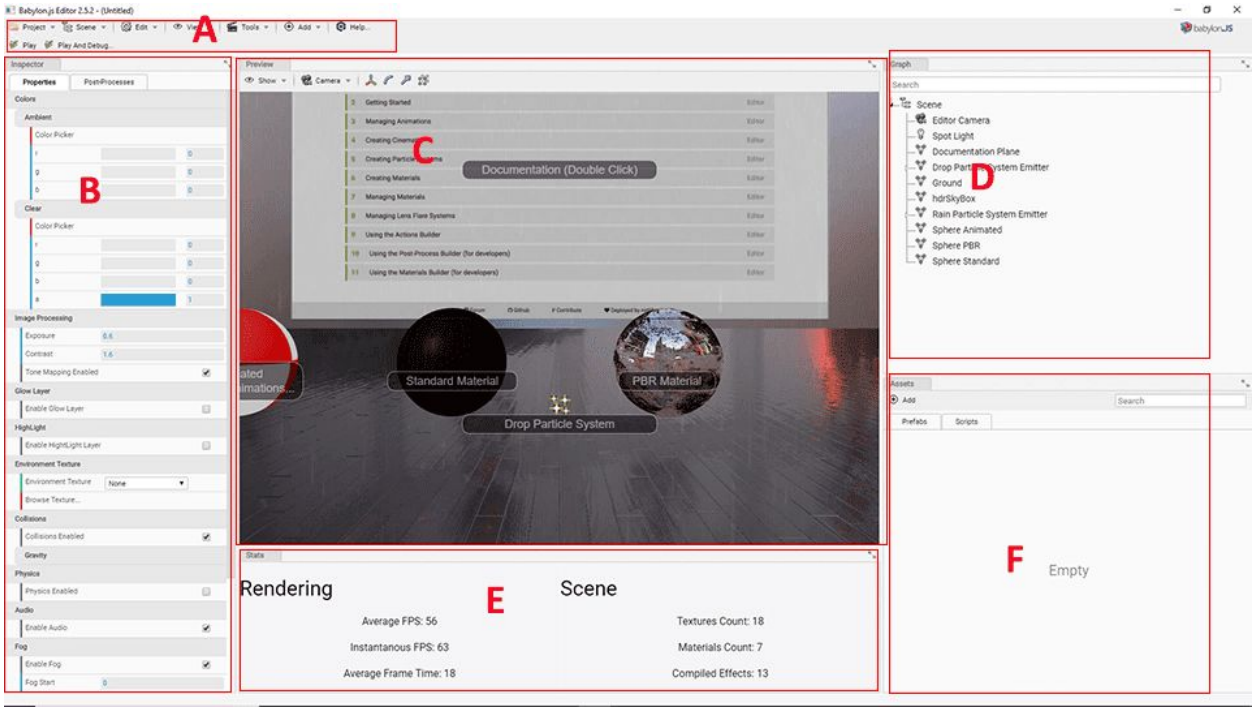
Installation is easy for the desktop editors; just download and install!

Babylon.JS Editor Layout

Before we jump into using the scene editor, we should clarify that we are working with a scene editor, not a full game engine such as Amazon Lumberyard, Unity, or Unreal. There are a lot of similarities, but the Babylon.JS editor is at its core a scene editor; allowing you the developer to quickly arrange meshes, add materials and textures, include scripting on objects, and manage the basic environment.

You will be able to use the environment to do a lot of the graphics side of development, but you should expect to do additional scripting to bring your project to completion.

For the next part, launch the Babylon.JS scene editor so that we can build something great!



The Babylon.JS editor launches the basic layout shown above:
A) Menu - providing access to additional resources and launching the Play and Debug window
B) Inspector - shows all of the properties for the scene. When no object is selected, the scene properties are shown. On some objects, multiple tabs might be present in the Inspector window
C) Scene Preview - shows the current configuration of the game scene. Includes scene controls at the top:

Show - controls the visibility of object bounding boxes, wireframe of meshes, Post-processes, Textures, and Lights in the current scene

Camera - Camera selection: currently limited to Free and Arc Rotate

 - Move tool - Allows movement of an object along the X, Y, or Z axes

 - Rotate tool - Allows rotation of an object along its X, Y, or Z axes

 - Stretch tool - Allows for the increase or decrease in the size of an object along the X, Y, or Z axes

 - Scale tool - Uniform scaling of a mesh

D) Scene Graph (labeled as Graph) - Shows all objects in the scene. An object can be selected from the Graph window

E) Stats - shows current scene stats such as Frames Per Second. **Code**, **Materials**, and **Textures** open as tabs in this window when selected from the **Tools** menu.

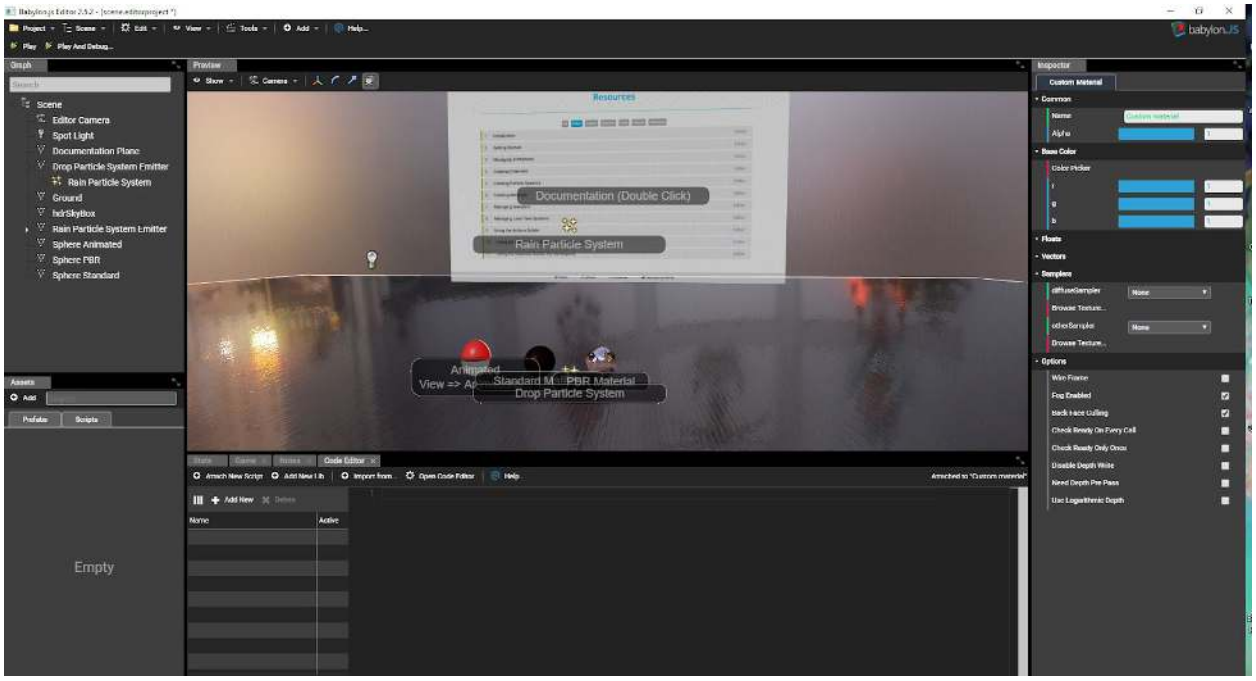
F) Assets - Assets (meshes, textures, scripts) available from the project folder to be added to the scene

Changing the Layout

The layout if fully customizable by the user. Any changes that you make will stay the same for the next time you open the editor.

To change the layout, click and drag the tab to a new location. Tabs can be combined.
You can also change to a dark theme by selecting **Edit** > **Dark Theme**

Below is the layout that I am currently using:



If you have used other game engines, the layout might look familiar.

Default Project

The Babylon.JS editor always starts by loading the default project. If you look closely, you can see that it has two particle effects (rain and raindrops), PBR (Physics-Based Rendering), Standard materials and textures applied, as well as GUI (Graphical User Interface) labels for each item and instructions. There is also a skybox, and ground applied to the environment.

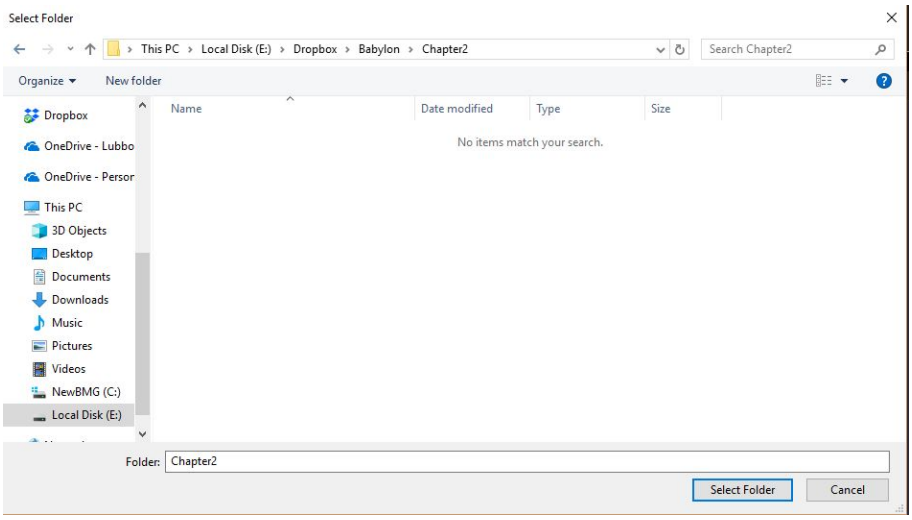
If you click on the Play button in the menu, a Game view will launch in the same window as the Stats, showing what the environment looks like when running.

Spend some time exploring the default project before we begin our own project.

Hello World In Editor

Now that you have some familiarity with the scene editor, let's start a new project. In the editor, click on **Project** > **New Project…**
You will see a dialogue box asking if you want to remove the current scene and create a new one? Click **Yes**

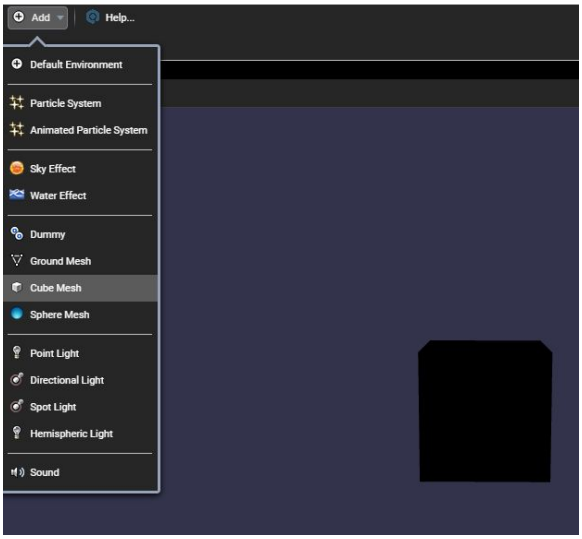
You should now have a blank scene with only an Editor Camera in the Graph.
First, let's set a location to save the new scene. Click **Project** > **Save Project**
On your hard drive, create a folder for the project. I created a folder called **Babylon**, then another folder within it called **Chapter2**.



And click the **Select Folder** button to complete the task.

Now, let's get something showing in the scene: Select the **Add** menu > **Cube Mesh**
This will add a box mesh, just like we did in Chapter 1.

As you can see, the cube is currently dark. This is because there currently isn't any light in the scene.



Let's add a light. Just as in the coding example from Chapter 1, there are four types of lights: Point Light, Directional Light, Spot Light, and Hemispheric Light.

Let's add a Hemispheric Light to the scene.

In the Inspector window Properties, you can adjust the cube's property, including its scene graph name, if it is enable, visible, if it has a material (color or texture) if it is a child of another object, its position, rotation, and scaling, as well as if it is pickable (which we will discuss later). You will also see that there is a second tab, called **Physics**.

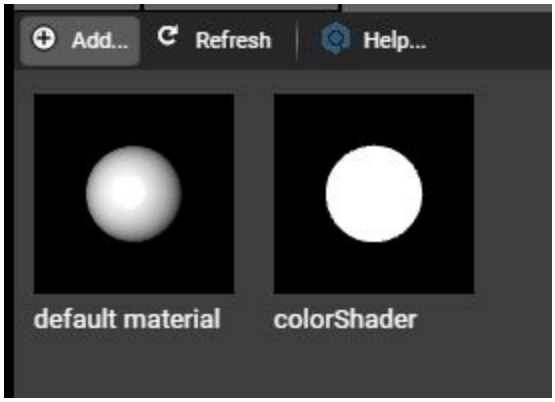
The physics tab controls how the object interacts with the surrounding environment. Can it collide with another object?
Right now, no, it can't, because there is not anything to collide with. Let's change that.

In the **Add** menu, select **Ground Mesh**. This will add a floor to our environment.
The Ground Mesh is a plane (two-dimensional object) and we will do more with it later.

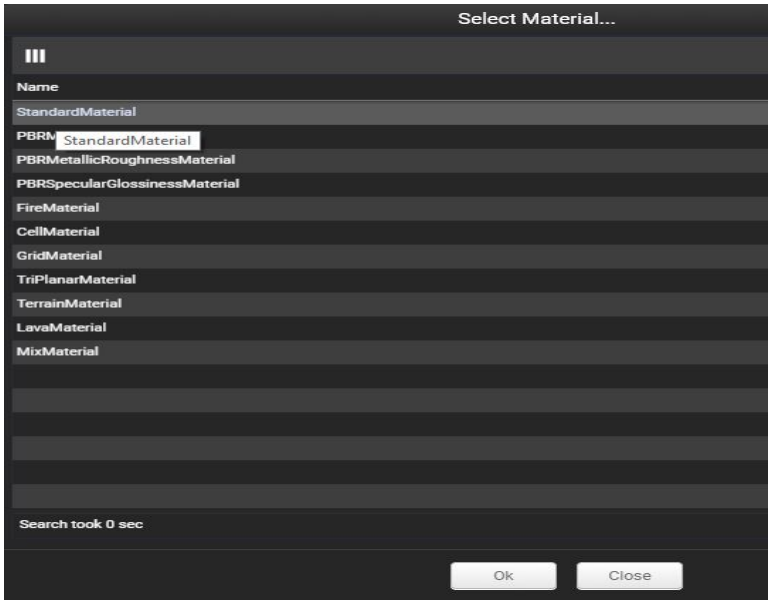
You might notice that the cube is intersecting the ground. Change that by clicking on the  control and the cube in either the Graph or the Preview. Click and drag on the green arrow to drag the cube above the ground mesh.

Now the cube is sitting in space of above the ground. Let's make it a little more visually interesting. Let's apply a color to the cube and ground so that it's easier to see (right now it is a polar bear in a snowstorm).

Click on **View** menu > **Materials Viewer**
This will open the Materials Viewer so that we can select or create materials. Click on the **Add** button above the default material.

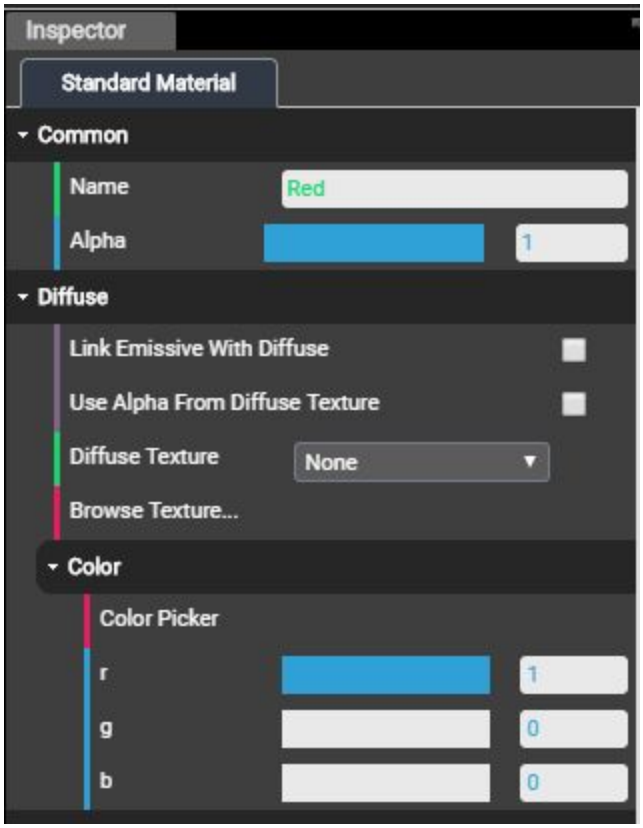


Next, select the **StandardMaterial** from the Select Material Dialog box.

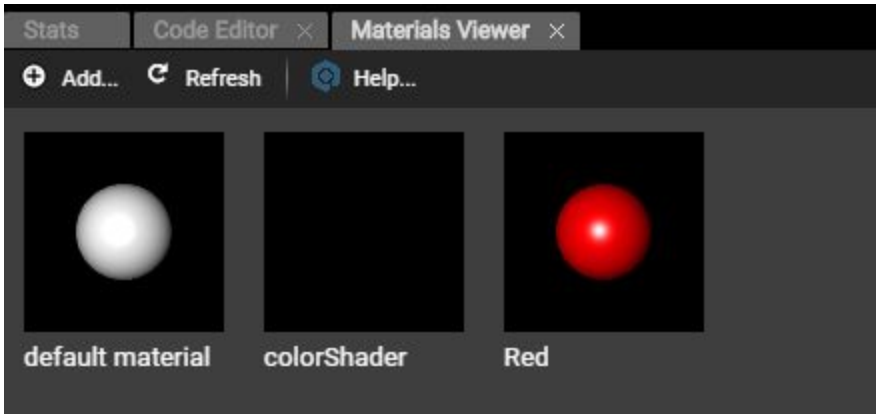


Click on the **StandardMaterial** in the **Material Viewer** and look at the **Inspector** properties. Change the **Name** to **Red**.

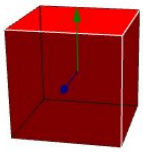
Then change the **Color** properties so that **g**(reen) and **b**(lue) are both **zero**. You can do this by dragging the blue bar to the left, or changing the values beside the bar.



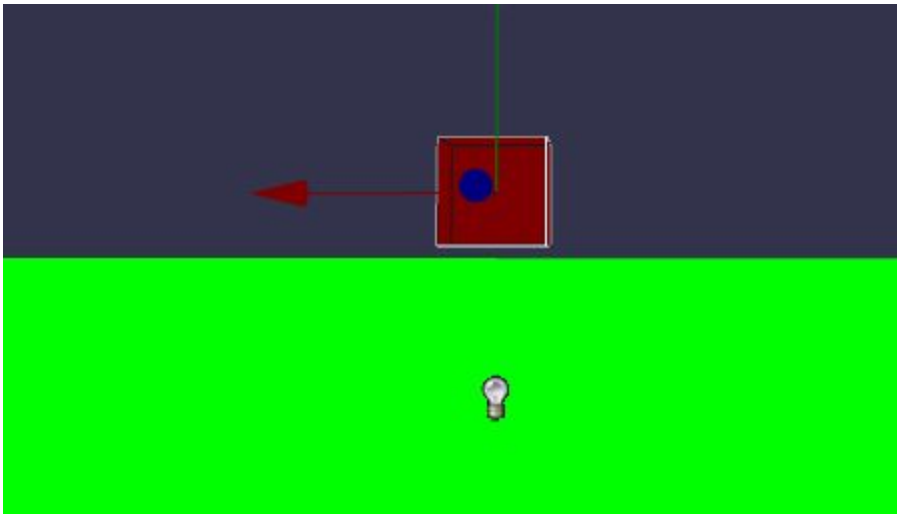
If the Materials Viewer doesn't update to show the red material, click the **Refresh** button in the Materials Viewer.



Drag the Red material on to the cube in the preview. You should now have a red cube!



Repeat the process to create a green standard material and apply it to the ground.



Now let’s have some fun and turn on physics!

Physics

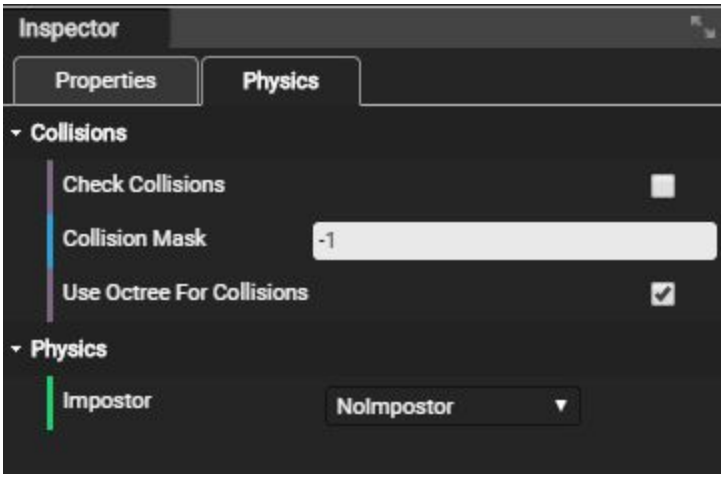
Babylon.JS allows you to select from several physics frameworks to simulate collisions, gravity, and interactions you would expect to see in a game environment. The editor uses the cannon.js and oimo.js libraries.

First, we need to turn on physics for the entire scene. To accomplish this herculean feat, click on the **Scene** in the Scene **Graph** window.

With the scene selected, scroll down the **Inspector** Properties until you see **Physics**.
Click on **Physics Enabled** so that the checkbox is checked.

Notice that **Gravity** is above Physics. If you click on the triangle beside Gravity, you will see the current gravity settings. It should start with the **Y** property set to -9.8. That means that gravity will pull objects down the Y axes, or toward the ground mesh. You might find that the -9.8 value a low for simulation of what you would expect. Crank the value up for objects to fall faster.

Now, select the **New Ground** object in the Scene **Graph**. With physics enabled, you will have a new resource under the tab in the Inspector for Physics. Click on the Physics tab in the Inspector.



Applying Imposters

While collisions were available before turning on physics, we also now have Physics, which allows us to set an **Impostor** for the ground.
What is an impostor you might ask?

An impostor is a stand-in or replacement for the mesh. It is a simple geometry that is used to calculate physics interactions within the environment.

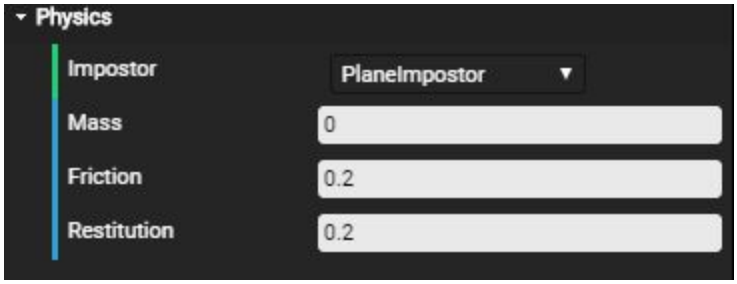
By default, objects in the environment do not have an impostor (thus the “NoImpostor”).

But, if we don’t apply an impostor to the ground mesh, objects will fall through, and keep falling.

Available Impostors to choose from include:

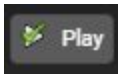
- No Impostor - Physics will not be applied to this object (default)
- Sphere Impostor - Use for sphere objects
- Box Impostor - Use for box or cube objects
- Plane Impostor - Used for a ground mesh. Can be replaced with a box impostor
- Mesh Impostor - Sets the impostor to the object mesh. Could significantly lower performance!
- Cylinder Impostor - Use of cylinder objects.
- Particle Impostor - Use for particle effects than need physics. Could lower performance!
- Height Map Impostor - Used for height map ground mesh

Select the Plane Impostor. When you select an impostor, you will get 3 more fields:



The **mass** set's the object's mass in the environment. If you set the mass as 0 (zero), it will not be affected by gravity. So if you want an object to be static or not fall, such as the ground, then leave the value as zero.
Friction is how much friction is applied to the object when it is moving along a surface. The default is 0.2. The higher the value, the more an object will be slowed.
Restitution is bounce. How much of the force of the object is retained when it collides and bounces. The higher the value, the more force that is applied.

Now that we have an impostor applied to the ground mesh, let's do the same for the Cube.
Select the **Cube** by clicking on it in the Scene Graph or in the Preview Window.
Select the **Physics** tab in the **Inspector** window.
Change the Impostor to BoxImpostor.
Change the **Mass** to **1**.

Click on Play 

in the menu bar to launch the Game window. The cube should fall and have a small bounce when it collides with the ground mesh.

There are a few other properties that you can control via the Physics tab in the Inspector window:

- Check Collisions - enables checking if two objects collide. Both objects must have Check Collisions on
- Collision Mask - restricts some types of collisions from registering
- Use Octree For Collisions - Useful if you have a lot of meshes (objects) in a scene. Speeds mesh selection by only registering meshes the camera can see

Saving Your Work

To save your work, select **Project** > **Save Project**. This will open the file dialog box and allow you to save your work to a folder. The Save Project (or Save Project As..) creates a scene.editorproject file and a folder names **scene**. The scene folder will have a copy of the scene file (as a .babylon file type), and any textures and meshes associated with the project. If you plan to load a mesh from another program such as Blender, 3DS Max, or Maya, the glb or

glTF file should be copied to the scene folder so that the editor has access to it, along with any associated textures. There is no such thing as saving too often.

The online editor (editor.babylonjs.com), the system will attempt to save the file to OneDrive (Microsoft's cloud storage solution).

Downloading a Scene

Under the Scene menu, you have the option to Download Scene. This allows you to create a scene that is not attached to a project, saving any basic meshes or the location of imported meshes in the scene. The default file time is .babylon. The Scene will not include project settings, lights, or cameras.

Saving a Project vs Downloading a Scene

Download Scene (under the **Scene** menu) will output a .babylon file of the scene, i.e. all of the meshes and materials in the scene.
Saving a Project (under the **Project** menu) will output a .editorProject file with the properties and changes made to the scene and scene resources.

Loading a Project

Using **Project** > **Import Project**, you can load your previously saved project to continue working on it. Importing a Project will replace the current project in the editor.

Import a Mesh

You can import meshes from a file (which we will discuss in the next chapter) using **Scene** > **Import Mesh From…** You should place any meshes and textures in the scene folder created when you saved the project for simplicity.

Publishing a Template

To publish your project, you will need to **Export Template** from the **Project** menu.

The Export Template command will:

1. Allow you to select the scene format (babylon, glTF, or glb)
2. Allow you to select the folder where your template will be saved. This should be an empty folder
3. Create the files needed to build the project (completed by Node.JS)

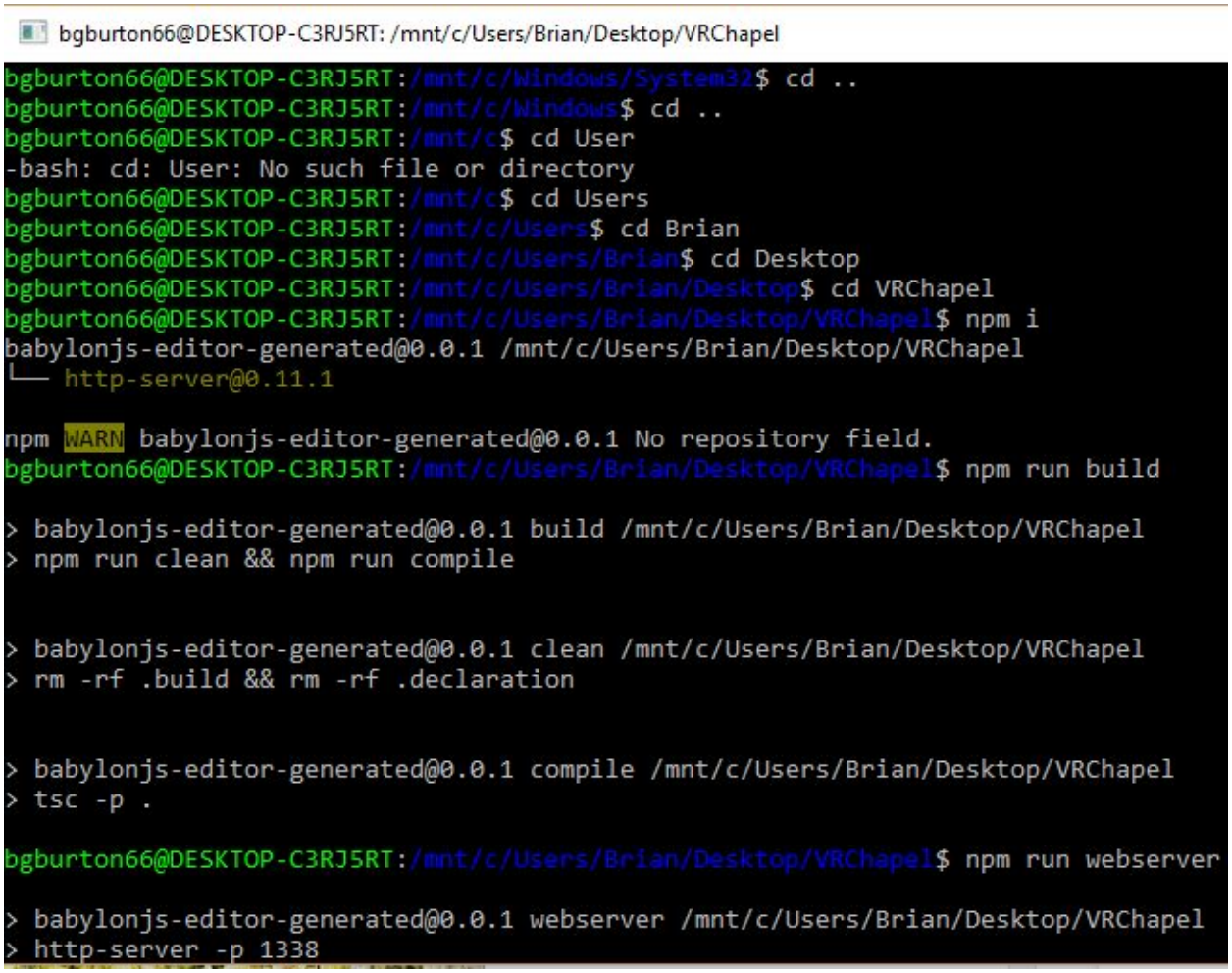
Files include in the template include:

package.json - a list of all install dependencies

index.html - for loading the project in a browser window
src folder - holds a test file to load your scene and the editor project
scene - has all the final assets and scripts for your project
READ.md - lists the available commands for product testing

Once you have the template, you are ready to build the project. This involves using Node.JS that you have already installed on your system.

From your Terminal (Mac) or Bash (Windows), navigate to the NEW folder that you created for the template. I have mine on the Desktop.



cd to the folder that contains the template
npm i
npm run build
npm run webserver

npm i will install the dependencies and modules needed for the project to run. Note that you may need to use **sudo npm i** to give administrative privileges to the installation process (needed to create folders on some operating systems).

npm run build reads the package.json file and builds the project by transcompiling the TypeScript files into JavaScript. It will also create the index.html file needed for your project. Once the build is completed, you can view the project locally by running the node webserver or upload to a web server to host your finished project.

npm run webserver launches a local web server on your computer so that you can see the project in a browser window. For local testing, you will be able to load your project a browser window using the URL http://localhost:1338/index.html

Working with a Team

Yes, you can work on a large (or small) project with a team of people. Using a shared network folder such as OneDrive, GitHub, Dropbox, or Google Drive, members of the same team can quickly contribute to the project. Using a version control tool will make life much more harmonious should someone on the team accidentally delete an important file.

One caveat: Only one person can edit a scene at a time. But if you have multiple scenes, that won't be a problem. Team members can also contribute models, textures, and sound to your scene.

Show the World

To place the project on a remote web server, upload the index.html file, the .build, and node_modules folder using your favorite FTP program (we use FileZilla) to the remote server or, if the server dashboard supports it, drag and drop the index.html, .build, and node_modules folder to the server.
We will address fine-tuning your project for a remote server in a later chapter.

For further information on setting up a remote web server, see the Appendix.

Summary

In this chapter, we introduce the scene editor, how to load meshes, and create standard materials. Then we proceeded to use physics and set impostors for the ground mesh and cube mesh. Once the mass was set for the cube, we ran the project to see physics in action.

Assignments

1) Change the restitution (bounce) value on the ground physics. How does this affect the impact of objects colliding with the ground?
2) Change the gravity setting by adjusting the x, y, and z values. What happens when you play?
3) Add a Sphere to the project. Load the Texture Viewer (View > Texture Viewer) and load a texture on to the sphere. Remember to copy the texture to the scene folder.
4) Create a blue cube and add it to the scene.
5) Publish your project and view it locally using a web browser.

Chapter 3. Adding Meshes and Interaction

Spector.js

Working with Meshes

glTF, GLB, OBJ

Working with Materials

PBR

Interaction:

Lens Flares

Particles

Adding Sound Effects

Adding Background Music

Summary

Questions

Assignments

Chapter 4. Adding Physics

Physics Engine

Imposters

Summary

Questions

Assignments

Chapter 5. Animation

An online conversion tool to glTF
https://blackthread.io/gltf-converter/

Basic Animation

Adding Animated Meshes

Summary

Questions

Assignments

Chapter 6. GUI

Keeping Score

Summary

Questions

Assignments

Chapter 7. Improving Performance

Summary

Questions

Assignments

Chapter 8.

Summary

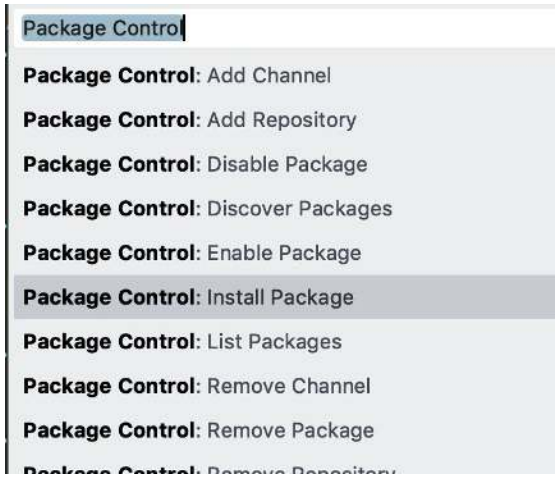
Questions

Assignments

Appendix

Installing TypeScript in Sublime Text Editor 3

Sublime Text Editor 3 is a popular IDE for coding. Thanks to its popularity, there are many plugins and packages that expand the functionality to speed up the development process.
Most plugins are handled via the Package Installer, which can be added via **Tools** > **Command Palette** then type **Package Control** and select **Install Package**



In the installation window, type in TypeScript and select the TypeScript package. This will add the ability to transcompile using the Build command in Tools.
Note that the transcompile function uses the local installation of Node.JS (covered later in the Appendix).

Recommended Extensions for Visual Studio Code

Using the Extensions tab  in Visual Studio, install the following extensions:

Babylon.js File viewer - will allow you to view glTF files in the folder that are < 5MB in size
Debugger for Chrome
Shader languages support for VS Code
Terminal - simplifies compiling of TypeScript files and provides terminal access in VS-Code

There are many other useful extensions available in VS-Code. Feel free to explore!

Install Node.JS for Local Testing

If you are going to test on your local machine with the editor, you will need to have Node.JS installed.

If you haven't installed Node.JS on your computer, follow the directions here:
http://nodejs.org/en/download/

It is recommended that you use Bash for your Windows terminal when working with Node.JS:
https://www.windowscentral.com/how-install-bash-shell-command-line-windows-10

There are two ways to run Node.JS for local web service:

Using Node.JS for simple web hosting

The **first** (used in Chapter 1) requires the installation of HTTP-server (you only need to do this the first time). Make sure that you have started Bash as an administrator (Windows) or Terminal (Mac).

1. Install HTTP-server

```
npm install http-server -g
```

If you get an error saying you need administrator rights then use:

```
sudo install http-server -g
```

2. Navigate to the correct folder. Note that Bash may show your directory structure in a different format than you are use to:

```
bgburton66@DESKTOP-C3RJ5RT:/mnt/c/Windows/System32$
```

The blue section shows the directory structure from a mount point (i.e, a Linux structure).
In my case, the folder with my Bayblon.JS files are located on a second hard drive, requiring me to navigate to the folder using cd (change directory):

```
bgburton66@DESKTOP-C3RJ5RT:/mnt/c/Windows/System32$ cd ..
bgburton66@DESKTOP-C3RJ5RT:/mnt/c/Windows$ cd ..
bgburton66@DESKTOP-C3RJ5RT:/mnt/c$ cd ..
bgburton66@DESKTOP-C3RJ5RT:/mnt$ cd e
bgburton66@DESKTOP-C3RJ5RT:/mnt/e$ cd Dropbox
```

Note that if your folder name has a space, you may need quotes around it: cd "My Babylon Stuff"

3. Once you have navigated to the folder, you can launch the server:

```
http-server ./ -p 1337
```

4. In your browser window, enter:

```
localhost:1337/index.html
```

Replacing index.html your webpage name.

NOTE: You can also use Apache, PHP, Python, or other local servers to handle simple web service. We have included the process for Node.JS since it is required for operations in later chapters. Of course, you can always just enter the file URL in the browser window too: file://c:/myhtmlfile.html

Using Node for Babylon.JS Editor

The **second** method is dependent on your using the Babylon.JS editor. When you publish the project template, you will need to navigate to the install folder, install supporting files, build, and run the web-server:

```
npm i
npm run build
npm webserver
```

You will receive feedback after each command. Once you complete the install and build, you will launch the webserver.
The project can be viewed in your browser:

```
localhost:1338/
```

Additional directions can be found on the GitHub repository for the Babylon.JS editor: https://github.com/BabylonJS/Editor

Setting up a Remote Web Server

You can set up your server on any Node.JS server. Since we will be using Node.JS, you can use a PaaS solution (Platform as a Service), which is generally very low cost or even free. We have provided links below to several popular PaaS services.

Amazon Web Services

AWS LightSail - https://lightsail.aws.amazon.com

Google Cloud

App Engine - https://cloud.google.com/appengine/

Microsoft Azure

App Service - https://azure.microsoft.com/en-us/services/app-service