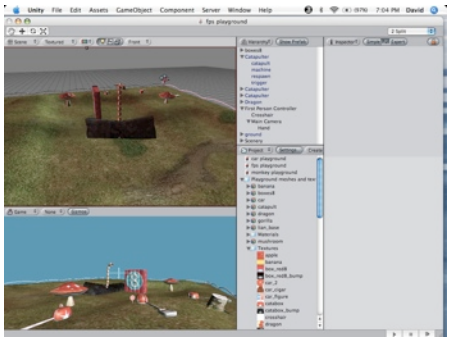


A Simple Unity Tutorial

A basic tutorial exploring the Unity engine, developed for new users to show some basic elements in the Unity engine: the Unity interface, creating worlds and objects, texturing, lighting, physics, game object control, and GUIs. 2006 by OTEE, David Janik-Jones, and his son, Cal. Version 3

Contents

Introduction

Over the Edge *really* have created a powerful game creation engine in their **Unity** application for the Mac platform.

Unity is fully capable of producing outstanding and complex game titles, as witnessed by OTEE's own shareware title Gooball (published by Ambrosia). One of Unity's strengths is that it provides this power in an accessible, well-designed interface whose workflow and methods are ideally suited to a small team of game designers. Unity provides a rich environment to develop games, and is as powerful and deep as the user needs it to be. Unity is extensible, emphasizes workflow, team asset management, allows for Javascript, C# or Boo programming, and is as good an engine as OTEE suggests it is.

The idea behind this tutorial is a very basic one. It is designed to provide a very quick and simple jump start into some of the basic elements and working methods of the Unity engine. It will provide new users with a comfortable starting point from which to explore more of the powerful features of Unity. It will show, in just a few steps, some of the remarkable basics that are part of Unity, and hopefully answer users questions about whether or not Unity is right for them.

Unity is very, very cool ... a game engine for Mac that is *everything* it claims to be and, for those with skill and determination, much more.

Thanks go to Joachim, Nicholas, David and Keli (some of the OTEE gang) for letting me participate in the beta-testing of their exciting product. A tip o' the forum posting hat to my fellow Unity users who have grown into such a great, knowledgeable and sharing community: especially Aarku, Marty, Robertseadog, Outcast, Richard_B, Kevin, Boxy, Paul and many others.

And to my gifted and wonderful eight year old son Cal, who came up with the idea for this tutorial and actually helped do some of what follows, much love from daddy.

The project files associated with this tutorial are available for download at http://www.otee.dk/tutorials/marble.zip.

Note

Unity Built In Help

Unity has a complete help reference built into the application. The right-most menu bar entry, as well as the small question mark to the right of the Inspector headings, gives you access to the Help at any time.

Hint

Breaking Up is Hard to Do

Panels can be split into two by clicking in the panel you want split and then selecting Window -> Split Vertical or Window -> Split Horizontal.

Putting them back together involves either right-click with your three-button mouse (see below), or CTRL-click on the divider line between the two panels you want to join. Then the command "Join" can be found.

Suggestion

Use a Three Button Mouse

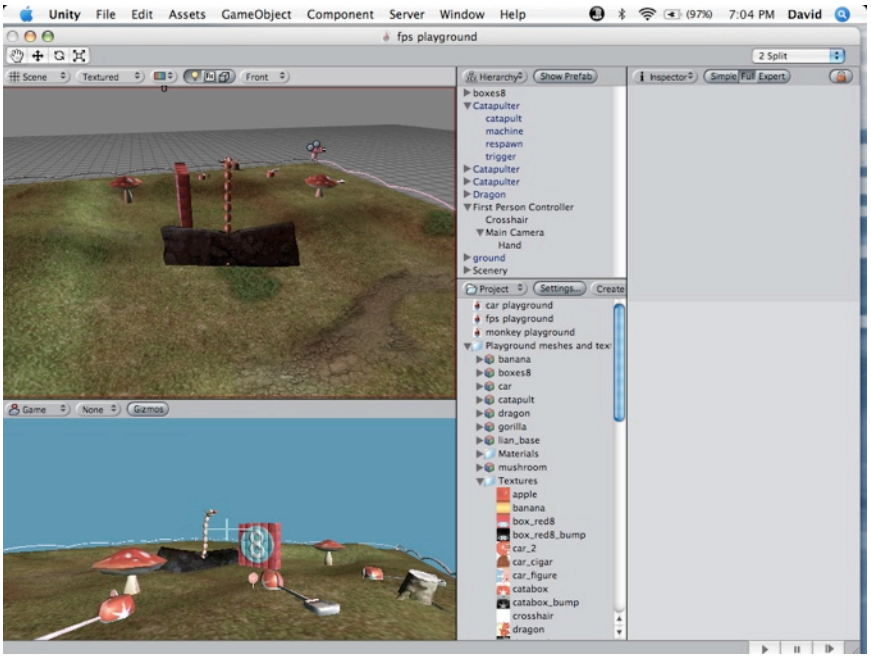
Even though I've been a Mac user since the days of a Mac Plus and am as die-hard a Mac fanatic as you'll find (granted, no rainbow Apple tattoos though), I'd highly recommend a decent three button mouse for use with Unity and any 3D-related application on the Mac.

It makes a number of tasks easier within the Unity environment.

If you're already using a 3D application like Maya, Cinema 4D, Lightwave, Cheetah 3D or Silo, you'll probably have one already.

I – The Unity Interface

Even though it's covered well in the Unity documents, I wanted to reiterate one or two things about Unity's interface. The interface is *customizable* and *savable*: how you like to work depending on your skill level, screen real estate, personal preferences, etc. can be saved as a layout. Layouts are saved and opened using the pulldown menu in the upper right of the screen.



I'd suggest that first time users and beginners work using the 2 Split layout, as shown above. The screen shot shows the 2 Split layout applied to the default demo project and scenes that are opened when you start Unity for the first time.

The 2 Split layout allows for any screen size to access the five main work panels. I find that these five panels are the main one's required to be visible and accessible during normal work: a Scene view, a (in) Game view, an Object Hierarchy list (showing objects in the game and their relationships to each other), a Project list (showing all of your assets; e.g., textures, objects, scripts, materials, etc), and the Inspector panel which shows details and allows editing of the currently selected asset.

These are the five things you'll always be working in and need to be most familiar with Unity, so it's a good idea to set it up this way to start.

Panels can be resized and split/joined as you like (see *Breaking Up Is Hard to Do* at left for important information).

Important!

What Happened to Unity?

The first time you see Unity "crash" and go away when you start a New Project you will freak out. I did.

Don't panic! It didn't crash!

Wait a few seconds and Unity comes back with your fresh new project ready to go.

Here's what happens ... When you create a new project in Unity, the whole engine needs to restart to create your project. This process takes a few seconds (around 5-10 seconds on my 1.5GHz G4 laptop).

I panicked when it looked like Unity had completely crashed back to Finder for no reason. In a few seconds though, you'll see Unity restarting and a progress bar appears that details how Unity is now updating your project's assets.

Whew. Scary when you first see it though, at least for this old guy.

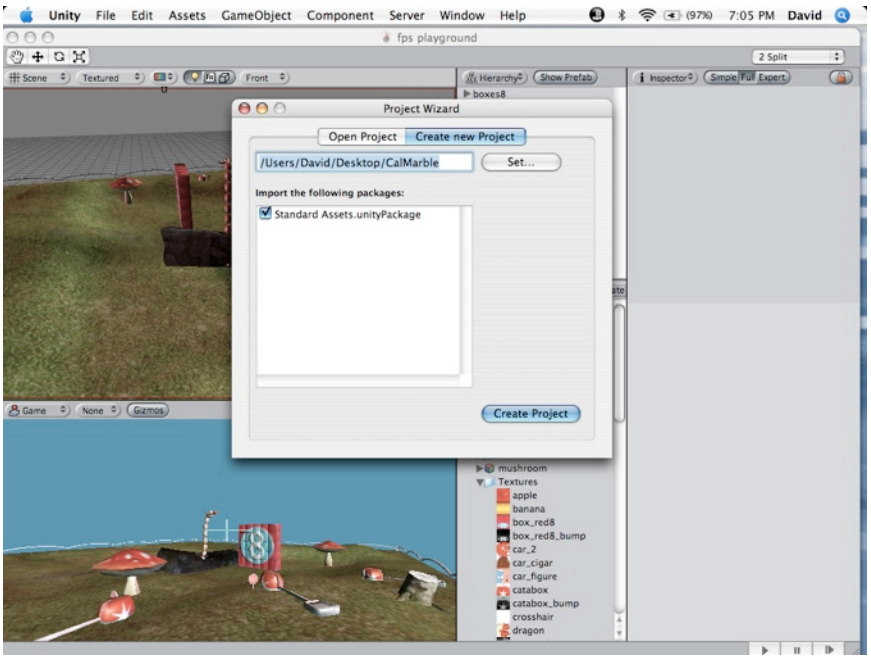
Hint

Naming Conventions

When creating a New Project, the last part of the path will create a directory/ folder with the project name attached. So /Users/Bob/Desktop/MyGreatGame will create your project in a folder called MyGreatGame on your desktop.

II – New Projects and Empty Worlds

Before creating your first new project, read the very important *What Happened to Unity?* note to the left. Go ahead, I'll wait.



Now that we're ready, and *aren't going to panic* when Unity restarts to create your new project, select File -> New Project... from the menu bar, as shown above. "Set" the pathway to your new project with the understanding that the last part of the path will be the folder name created for your project (CalMarble in the example above).

Now wait for a few seconds while Unity restarts and creates your new project. What Unity is doing during this time is: creating your project folder, writing into your project folder a bunch of standard assets and Unity stuff. The standard assets include many useful textures, materials, a basic font, lights, particles, scripts and other useful goodies. I assume that as Unity matures even more, and incremental versions get released, the OTEE gang will include new and/or improved standard assets in the Unity package.

Assets like scripts, models, and textures that other Unity users create and modify will also be made available on the public, and very helpful, Unity forums at http://otee.dk/forums.

That's now done! You've now created a complete game world that has much of the hard stuff (physics etc) done already!

Try This

Changing Your World

Notice how the Game view shows a blue background? Don't like blue and feel like making the first change in your world? Here's how to do that...

Click on the Main Camera in either the Scene panel or Hierarchy panel. Look at the Inspector palette and click on the Back Ground Color attribute to change the background colour.

Screen Captures

SHIFT-COMMAND-3

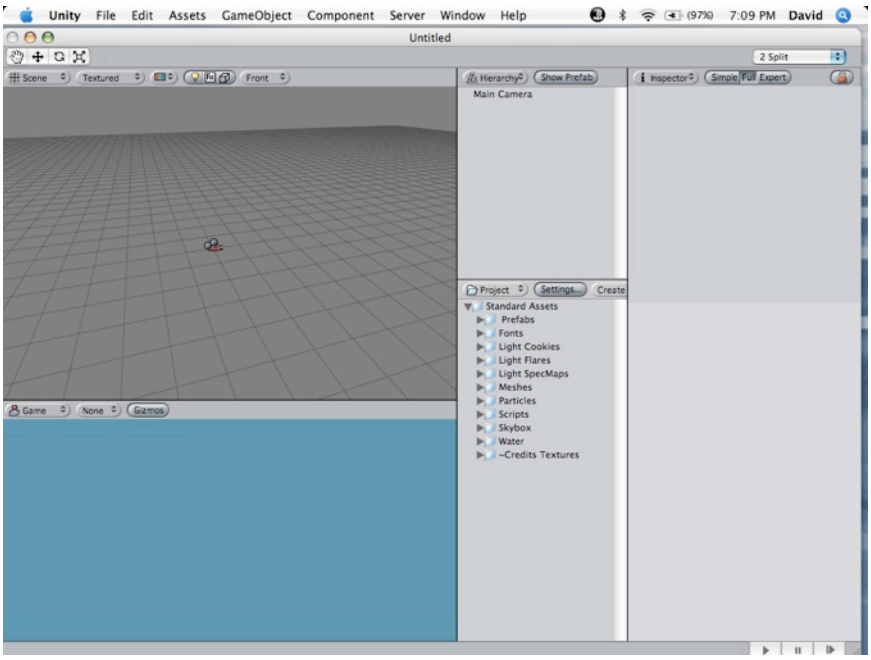
You can take screen shots of your work in Unity by using the standard key-board combination of SHIFT-CMD-3. The images will be saved to your desktop as either a PDF (10.3) or a PNG file (10.4).

Have A "Real" Mouse?

So You Have A 3-Button Mouse

Like many standard 3D applications, Unity works best with a three button mouse. To move around the Scene view with a three button mouse: holding down the ALT key with left button orbits, with middle button pans, and with right button or scroll wheel zooms.

You'll now see your empty world as shown in the next screen shot:



The Empty World

OK, so there's your world. An empty world with a single camera (visible in the Scene panel above). Select the Scene panel and notice it's slightly highlighted with a red, single pixel border. Swing the view around (*orbit*) the scene view by holding OPTION down and clicking and holding the mouse button.

To move around the Scene view using a one button mouse make sure to have the View tool selected and do the following: OPTION and mouse click-drag *orbits*, click-drag without a modifier key *pans*, COMMAND and mouse click-drag *zooms*.

If you have a three-button mouse, read *Have A "Real" Mouse?*.

The panel under the Scene view is the Game view from the one main camera and currently shows nothing. That's because there is nothing yet in the world you created.

Click on the camera itself in the Scene panel, or select the camera by clicking on the Main Camera label in the Hierarchy panel. Notice the "*manipulate object*" buttons in the top left now has the move symbol selected and your camera has the XYZ axis move arrows showing. Feel free to move the camera around, or click on the other "manipulate object" buttons to *rotate* or *scale* the camera (scaling has no effect as camera "size" doesn't matter).

Important!

Making Sure Your Changes Stick

Any time you make a change to one of your game asset's attributes or value within the Inspector panel, you'll want to hit the RETURN key to make sure they're entered.

Also note that changes made will not be permanent if you are in "Run/Play" mode at the time.

A Unity Strength

Create and Test In The Same GUI

One of the great strengths of the Unity engine is that you can create and test your game world in the same application, without the need to actually compile a standalone application to view the game. Use the Play button in the lower left of the Unity interface to run and test your current scene.

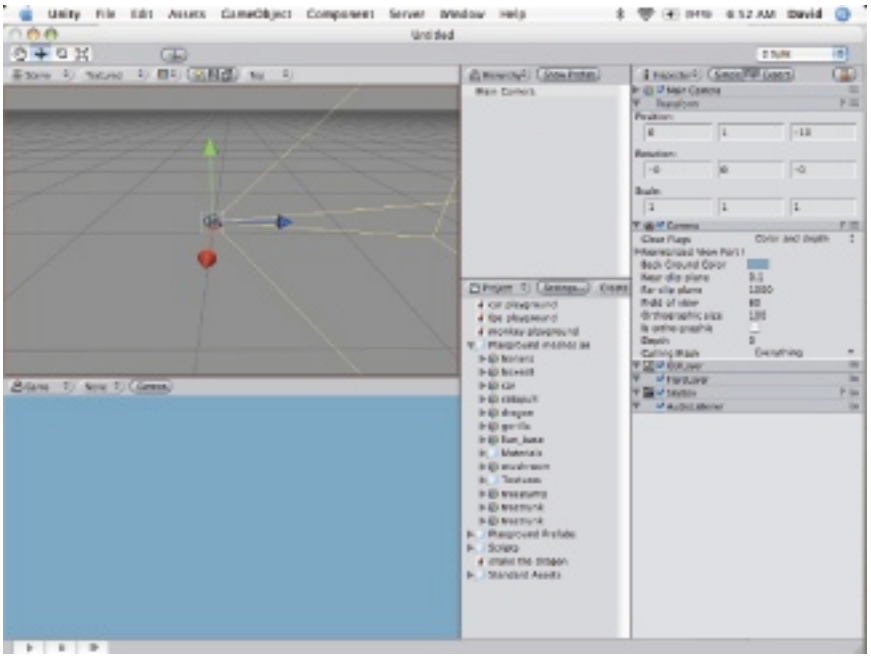
Try This

Making A Panel Full Screen

You can easily make any of the panels in the Unity interface full screen. Simply click in the panel you want to be full screen (notice the red, one pixel border highlight) and click the SPACEBAR. When you want to go back to your normal interface, click the SPACEBAR again.

This is especially helpful when moving objects around in the Scene panel, or for testing and viewing the game in the Game panel while in Play mode.

The Inspector panel (on the right) has also been updated to show details of the currently selected object; the camera in this case, as shown in the following screen shot.



The middle bottom panel in this screen shot is the Project list and shows the standard assets that Unity builds into your new project. Take some time to browse through the list of assets that the Unity engine provides. Also notice how clicking on one of brings up detailed information about them in the Inspector panel.

If you're not familiar with this panel, take a few minutes now to explore the standard assets and look at their associated settings in the Inspector panel.

Those Three Buttons in the Lower Right

Notice the three buttons in the very lower left of the Unity window that look like a VCR control? These Control buttons actually allow you to change the state of your project from *edit* mode to *run*. The three buttons are: Play, Pause, and Step. The keyboard equivalents are CMD-P, CMD-SHIFT-P, and CMD-OPTION-P. By clicking the Play button, you actually can watch your game running, in both the Scene and Game panels, and actually make **non-permanent** adjustments to object attributes in the Inspector panel while the game runs. To get back to editing mode, click the Play button again.

Save your project's first scene now by using File -> Save as ... and save it in the default location (Assets folder).

Note

Projects Versus Scene

In Unity terminology, a game is a "project" that consists of the stuff in the game (assets), and "scenes" that make up the various intros, cut-scenes, and game levels.

Your project can consist of any number of unique scenes, with the opportunity to make transitions from scene to scene happen as required. Scenes are shown in the Project panel as white icons with a small cube icon in them.

Another Unity Strength

Unity Recognizes Lots of Formats!

The assets you add to your project can be in a wide variety of file formats. These formats include such formats as: PSD (Photoshop) files, TIFF, JPG, TGA, GIF, PNG, BMP, IFF, PICT, multi-layer PSD, multi-layer TIFF, Wav, aiff, ogg vorbis, mp3, fbx, Maya mb/ma, 3ds, Blender, Javascript, C#, etc.

Multi-layered PSD and TIFF files are flattened on import, but remain layered and editable to the designers and team.

New assets are added in the simple manner outlined at right ... just add them to the appropriate directory within your "Working Assets" folder. Then they appear in the Project panel ready for use!

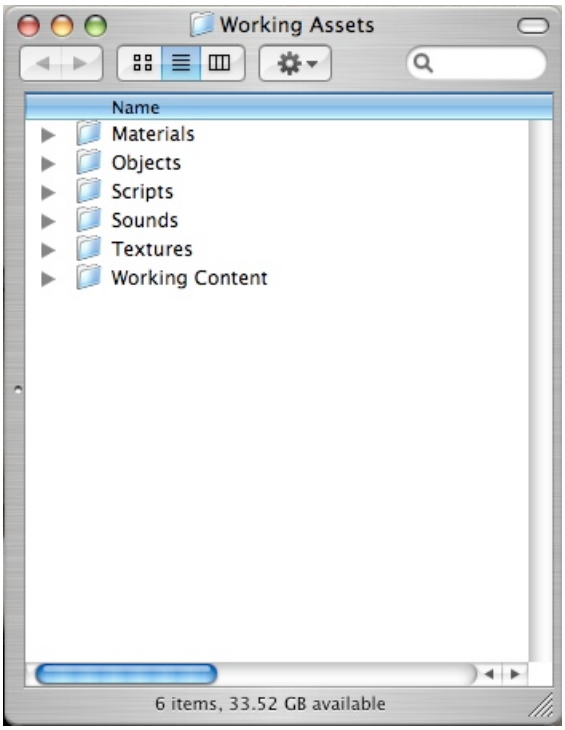
III – Scenes, Assets and Workflow

Now that you've saved a scene in your first project, we can get to adding stuff to the world you've created. Unity has added many basic standard assets to your project, but for good workflow practices we need to create a new place where you can add your own, project specific assets.

I'll explain my reasoning for this as it applies to the tutorial below.

It doesn't matter if you've quit Unity or it's simply hidden or in the background.

First, create a folder on your desktop called "Working Assets" and create subfolders in it as shown in the screen shot below. You can modify the names and/or structure of these subfolders in any way you'd like, but for tutorial purposes this is a logical, basic structure for new users that organizes and contains most of the types of project specific assets you'll be creating and adding to your game world.



Second, add this folder to your project by simply dragging the Working Assets folder into the "Assets" folder in your main project folder. That's it. These folders (and any contents within) will be automatically added to your project.

Suggestion

Work How You Want

I am suggesting for these naming schemes and work methods for this tutorial as a guide only. I've found these methods work best for me, as a beginning programmer, and many beginners to Unity, but Unity's strength is that you can customize it to the nth degree.

Create and save panel layouts that you like. Create your project specific asset folders and name the subfolders whatever makes sense to you. Customize the keyboard commands and shortcuts to your liking.

Get to know Unity and you'll be able to make it a very powerful tool customized to exactly how you want to work with it.

Another Unity Strength

Tools For Everyone's Budget

In an ideal world, everyone would own the latest copy of Creative Suite 2 and Maya Unlimited. We'd also own copies of Cinema 4D and Lightwave for good measure, all running on a 2.5GHz quad G5 with 8GB RAM and 30" displays.

But Unity accepts such a wide range of assets (objects and textures etc) that even game designers on a budget can find tools that will work.

For 3D work, there's Blender (free), as well as Martin Wengenmayer's excellent shareware Cheetah3D (Unity imports .jas files directly), and Silo by Nevercenter (a fast polygon modeling app).

For images, GIMP and Photoshop Elements can export files for use in Unity. For editing script files there's the free TextEdit from Barebones and the very highly recommended SubEthEdit by TheCodingMonkeys.

The cool thing? You add your own project specific assets like models, textures, materials, scripts, sounds etc in *exactly* the same way! Just drag your newly created asset file into your own Working Assets directory subfolder and they appear in your game ready for use!

Important Note: This asset structure is recommended **only** for small sample projects like this tutorial. Grouping assets by their ***function*** rather than *type* within larger game projects makes it much easier to locate an asset; i.e., directories like Ball, HUD, Levels, Menus, Props, SFX, Scripts, etc. with subdirectories make more sense for larger projects and teams.

My Rambling Ideas About Workflow

I've suggested using a 2 Split layout when starting in Unity and also how to best organize your *tutorial-sized* project assets folder to ease new Unity users into the program. Feel free, however, to experiment and set up Unity in whatever way best meets your workflow needs.

About workflow — Unity is a great tool in that it was designed from the ground up to be a game creation tool with a Mac aesthetic and with a Mac workflow method in mind. Unity permits a small group of game programmers to create great games by streamlining the process and making how each part of creating a game flows into each other *really* function.

Game assets like objects, textures, and scripts can be worked on at the same time as the game itself, and updated versions of these can be refreshed in the Unity interface with a single command. Visual assets can be produced using a wide range of applications and formats. This means you can have one person updating textures in Photoshop, while another writes scripts, while yet another works within Unity to refresh the assets, assemble all of these pieces together, and test how things are looking and functioning.

This workflow is ideal as many things can get worked on while maintaining the integrity and scope of the project through the Unity engine.

Now that we've had enough philosophy ... let's create some content and get our game off the ground!

Hint

Level = Scene

Since this tutorial only represents a single level of game play in our "marbles" game, there will be only a single scene in the project. In more complex games, you'd probably want to create each level as a scene, and also create scenes for cut-scenes, intro movies, GUIs, etc.

Transitions between scenes can be triggered by anything you can script such as collisions, variables like score, object position, etc.

Note

Textures X 2

Remember to save any textures that you create and want to import into Unity in powers of 2. But**,** the length and width do not have to be the same. In other words, all of the following textures sizes are OK:

64x64, 128x256, 1024x256, 512x64, 256x512, 1024x1024, 128x128.

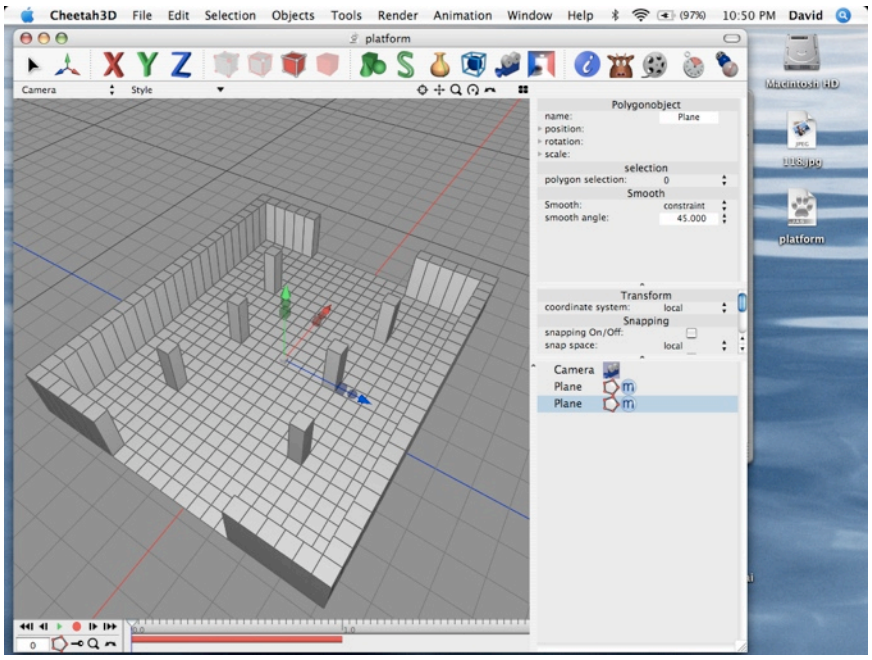
IV – Creating and Texturing Objects

This section isn't meant to be a lesson in how to create 3D objects; hopefully you (or someone who you bribed to make stuff for you) knows how to do that.

We want to create content for our tutorial. The game *concept* for our tutorial is this: There are a number of increasingly complex/ tricky levels (platforms) on which your player (marble) has to travel through and knock the other (enemy) marbles off in the least amount of time possible. Our tutorial will create a platform, player and enemy objects, apply physics, add a particle system, and player control on a single one of these levels.

In the hypothetical *complete* game, your marble would gain points by clearing the other marbles off as fast as possible, and gain bonus points by gathering "special" marbles, while generally avoiding falling off or out of the platforms.

The screen shot below shows the level my son wanted built, created using Martin Wengenmayer's excellent 3D shareware application Cheetah3D.



This level consists of a flat platform with some pillars as obstacles (to bounce off) and opening to knock the other marbles off (or fall off if you're not too careful). The only thing missing from the model is the texturing ... which is shown in the screen shot below.

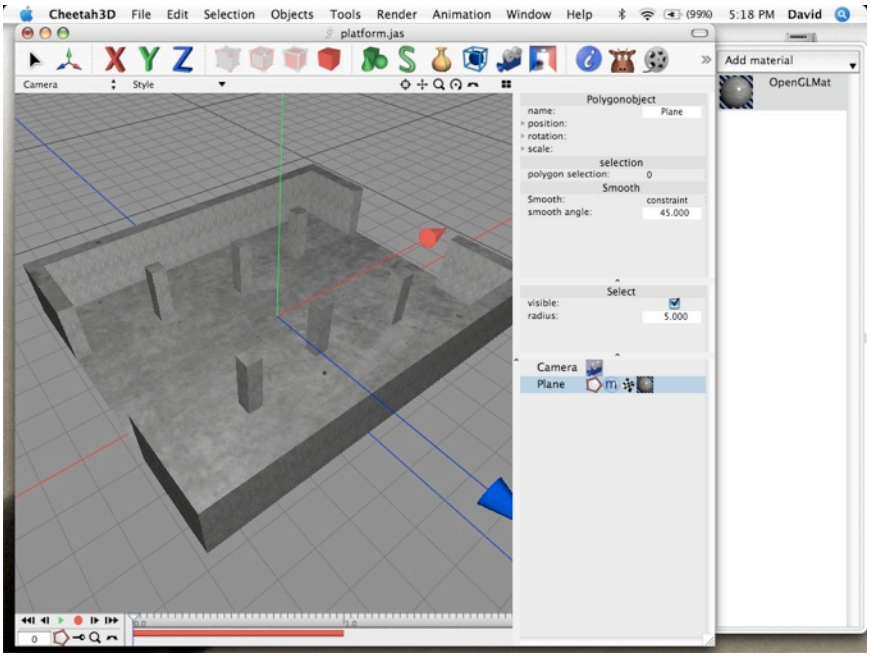
Textures In Unity

UV Mapping And The Like

You can texture your 3D game objects either within the Unity engine itself or in your 3D program.

My own preference is, for the most part, to texture in your 3D application and import it directly into Unity already textured.

You can stick a texture in a "textures" directory next to your exported .jas model, and Unity will find it for you.



Here is the model with a UV texturing (a JPG file) that looks like concrete applied. It's important, at least when using Cheetah 3D, to use the *UV texturing mode* because when you bring the model into your project Unity offers UV as the only option for textures and will bring it in exactly as you built it in your 3D program, textures wrapped correctly.

Getting this model into Unity is very simple. Getting this model into Unity is very simple. Simply save the .jas file and place it in your project specific Assets folder. Place the texture you used in your 3D program into a "Textures" directory next to your object. Your new asset will automatically update in the Project list. That's it, one platform created and in your game world.

If you use other 3D applications to create objects and meshes, you can find instructions about importing them online at: http://www.otee.dk/Documentation/Manual/HOWTO-importObject.html

It's now time to add the object we've just imported into the Scene panel, add lights, the player objects (marbles), cool particle effects, set up physics, and player controls.

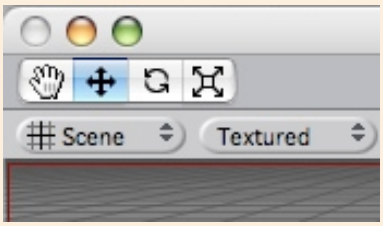
Important!

My Object Is Upside Down!

Depending on the settings in your 3D application of choice, objects may not be oriented and/or scaled correctly when dragged into the Scene panel from the Project list panel.

The easy fix is to select them by clicking on the object in the Scene view, and change the variables (scale, rotation) in the Inspector panel.

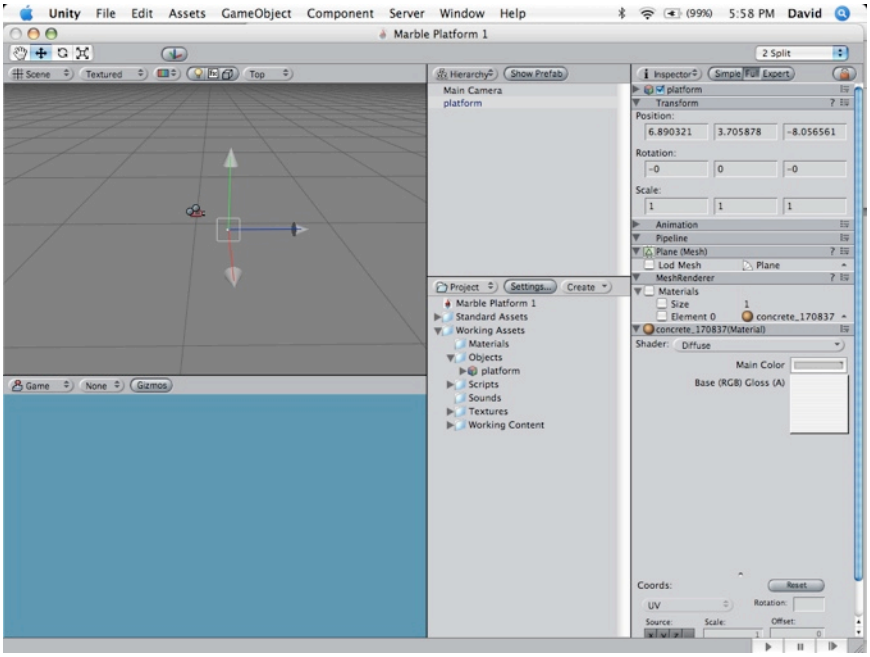
You can also interactively make changes to the objects' scale, rotation, or position, by using the "manipulate objects" tools in the upper left corner of the Unity interface as shown below.



Or use a combination of both (the rotation manipulation tool until it's almost 90° then use the Inspector to change the 86.77654 value you got by manually rotating it to 90).

V – Getting "Into" the Game

First, we want to place the platform we've imported into our project into the actual game by dragging an instance of the platform in our Project list panel out into the Scene panel. Your Scene panel should now look something like this:



Notice the incorrect scaling of the platform in my project? Using the techniques outlined at right in *My Object Is Upside Down!*, we can scale the platform object up. Your scene will now look like the screen shot shown at the top of the next page.

Notice the information about the platform is now showing in the Inspector panel, including the fact that we've scaled the platform object by 50 along all three axis. The Hierarchy panel (that shows the objects actually in the game scene) now also shows both a Main Camera and a "platform." We can rename the "platform" object by selecting it and giving it a new name in the Inspector panel (at the top of the panel).

In the screen shot, we've also changed the position and angle of the Main Camera object so that it's pointing down at the platform surface. We can also change the "*Far clip plane*" and "*Field of view*" value in the Inspector panel to focus more on the platform (to 500 and 45). Change the values contextually by CTRL-click (or right-click) and dragging. Hold SHIFT as well to change the values faster.

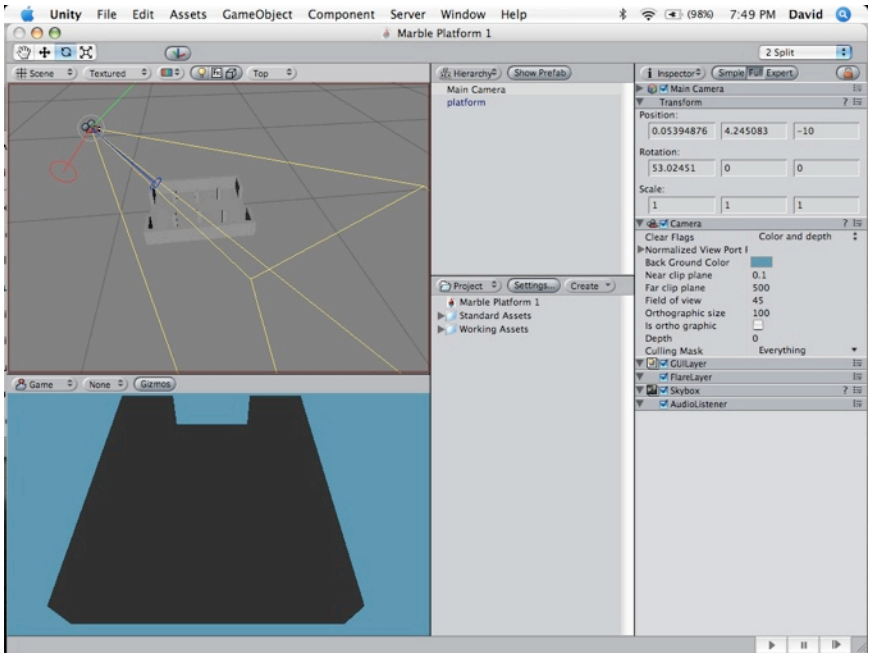
Note: In the Game panel there is no light on our surface yet? We'll add a light to start the next section of the tutorial.

Try This

Use Unity To Add Primitive Shapes

Unity has the ability to add a variety of objects to the current scene, among them several primitive shapes: cubes, spheres, capsules, cylinders and planes.

For this tutorial I'm going to use Unity to add something as simple as our marbles (just a sphere) to the game, rather than create them in a 3D package. I'm doing this to later show how to use Unity to apply textures to them.



Note

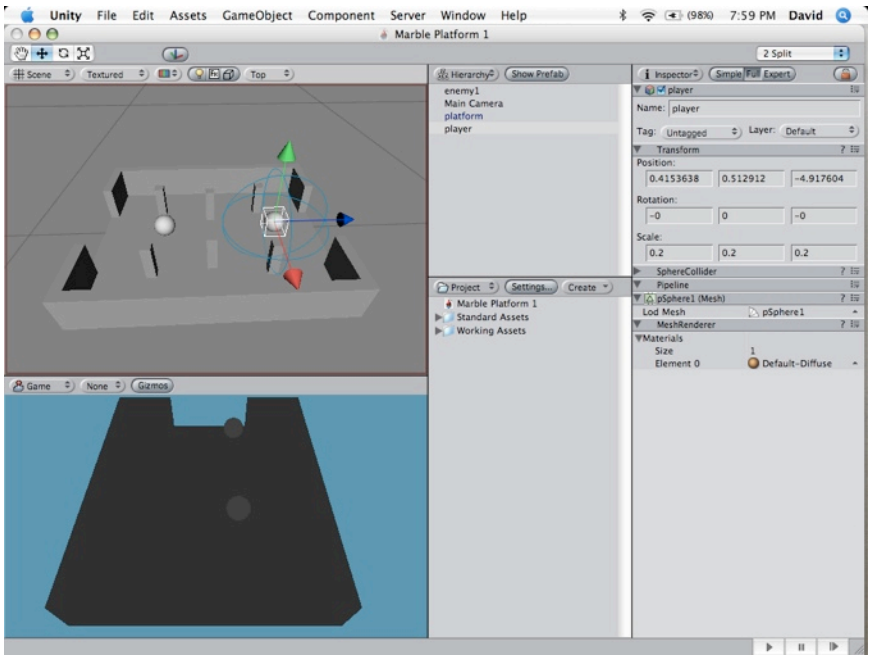
Save. Save. Save.

As a senior graphic designer, I've gotten into the habit of saving what I'm doing practically any time my hands stop moving (if you've ever lost 15 pages of financials in an annual report because the power fails you know what I mean). You should be saving your Unity project frequently; certainly at the beginning of each of these sections and just before you try something new. And whenever you can at other times; better safe than sorry.

Make sure to save your tutorial now.

Go ahead, I'll wait.

Let's finish by adding two spheres to the scene; one will be the player controlled marble, while the other will be one instance of an "enemy" marble. Select GameObject -> Create Other ->Sphere in the menu bar and create two spheres. Click on each of them, use the Inspector panel, scale them to 0.2 along all axis, rename them "player" and "enemy1", then use the *manipulate tools* to move them to be slightly above the platform as shown below:



We're now done adding objects into our project's Scene panel. Let's move on to lights, particles, and basic properties.

Textures In Unity

UV Mapping In Maya (Reminder)

If you stick a texture in a "textures" directory next to your exported .3ds model, Unity will find it for you. But the texture name needs to conform to 8.3 DOS file naming conventions (an affect of using .3ds as your file format).

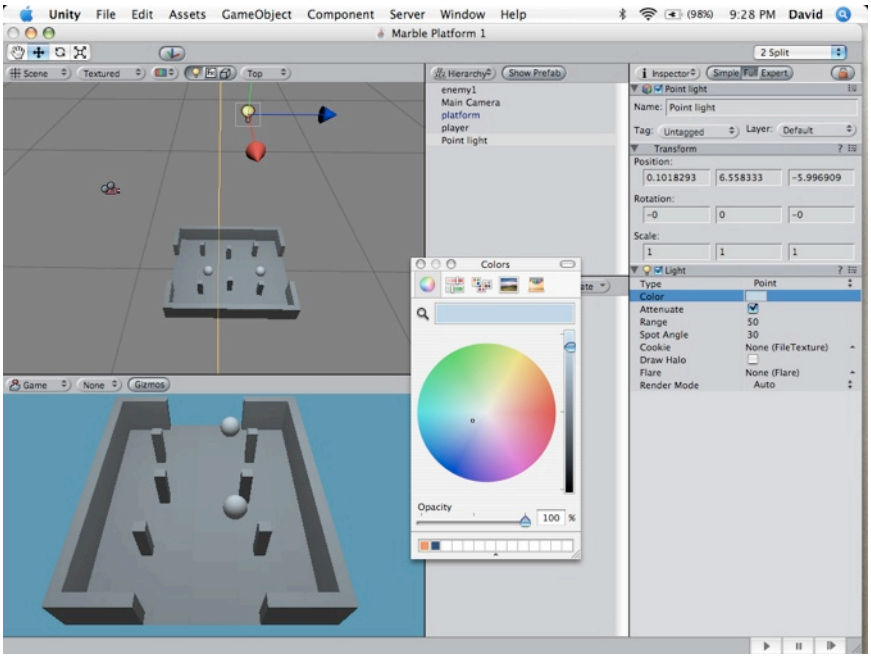
Maya is the recommended 3D content creation tool for use with Unity.

VI – Lights, Cameras, Particles, Action!

Let's finally add a light to the project. Lights come in the three standard types: point, spot, and directional. We're going to add a simple point light that will illuminate the entire scene we're creating.

The easiest way to do this is to select GameObject -> Create Other -> Create Point Light from the menu bar. This will add a new light to your Scene panel. Position it slightly above the middle of the platform.

Now change the light's attributes using the Inspector panel. Change it's *Range* to 50, and experiment by changing the light's colour. The results look like this:



Notice how the Game panel now shows our game lit by this light? Save your tutorial, and then feel free to experiment with the light by changing it's type, colour, angle, range, etc.

We *may* need to **fix** something at this point. Notice how our platform has no texture; it's because we didn't follow the .3ds texture naming conventions (see *Textures in Unity* at left).

We didn't notice this until we brought the light into the scene. Fix this by re-linking the object within the Scene window with texture we used to create the model. To do this, simply drag the texture asset from the Project panel onto the object in the Scene panel as shown below:

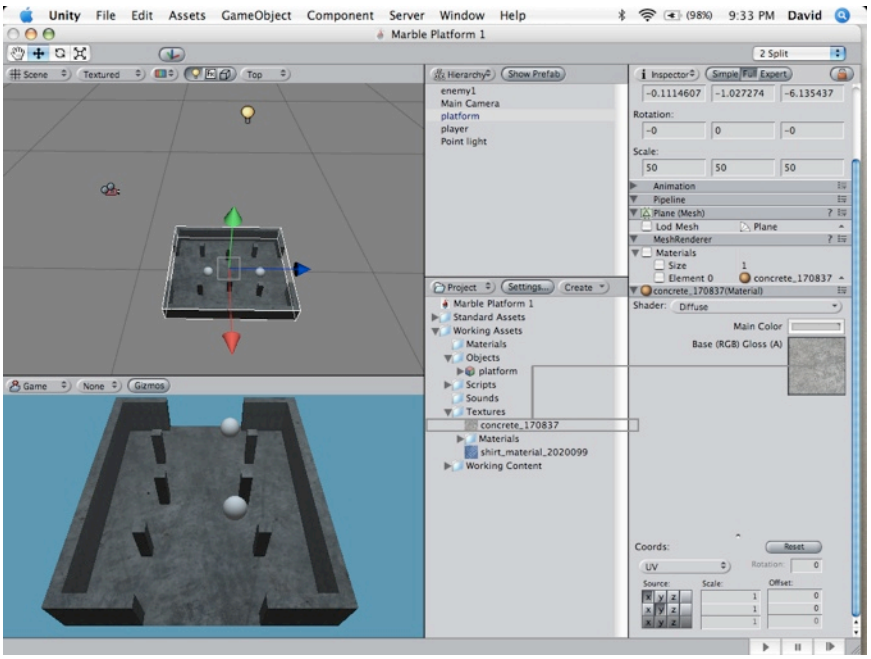
Future Features(?)

Aiming and Windows

While future features can't be guaranteed, the folks at OTEE do listen to their users and implement changes and new features when the requested feature(s) add value to Unity. The Unity forum is the place to mention these, as well as get advice and share scripts, objects etc. Two features should be noted as (possibly) forthcoming.

First, a camera (and light?) "aiming point" feature. This will enable a user to place small, non-visible target object into a scene that can be linked to a camera so that you can move this object to aim a camera, rather than having to fiddle with the camera itself to aim it.

Second, the panels (or panes if you prefer) may use a method to close and split etc. more like the application Final Cut Pro.



Now the platform has the correct UV mapped texture in both the Scene and Game panels.

Let's also, as an experiment, add some *particle systems* to the level we're creating. In this tutorial these are **solely** to add atmosphere to the level and to illustrate the basic techniques used, rather than serving any specific in-game purpose. Unity has the ability to add trails, particles, fire, smoke, and a myriad of other effects to the game environment.

To do this, select Game Object -> Create Other -> Create Particle System. The using the *manipulation tools* move the newly created particle system object slightly above and over the middle of the platform in the Scene panel. Then use the Inspector panel and modify the particle system as follows:

a) change the Minimum Emission to a value of 20;

b) change the Maximum Emission to a value of 30;

c) change the ellipsoid X value to 2 and Z value to 3 to stretch out the particle field to cover the platform; and

d) under ParticleAnimator change the Y force to -0.5 and the Damping to 1 (to cause the particles to fall downwards).

As an experiment, change the animated colours as well. The following screen shot shows our current project with the new particle generator:

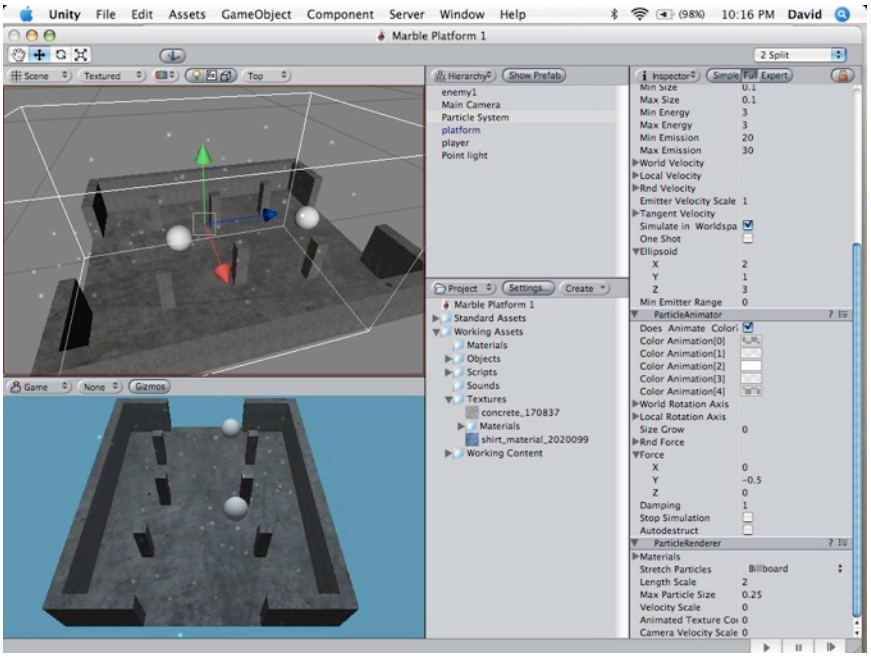
Experiment

Poke Around The Menu Bar

If you haven't done so already, take the time to poke around and explore the various menus and submenus in the menu bar of the Unity application. Some of the stuff will be familiar and intuitive, while other functions will leave you scratching your head.

If you see something interesting, save your project and give it a try (e.g., create a new camera or light, tinker with the project setting, etc).

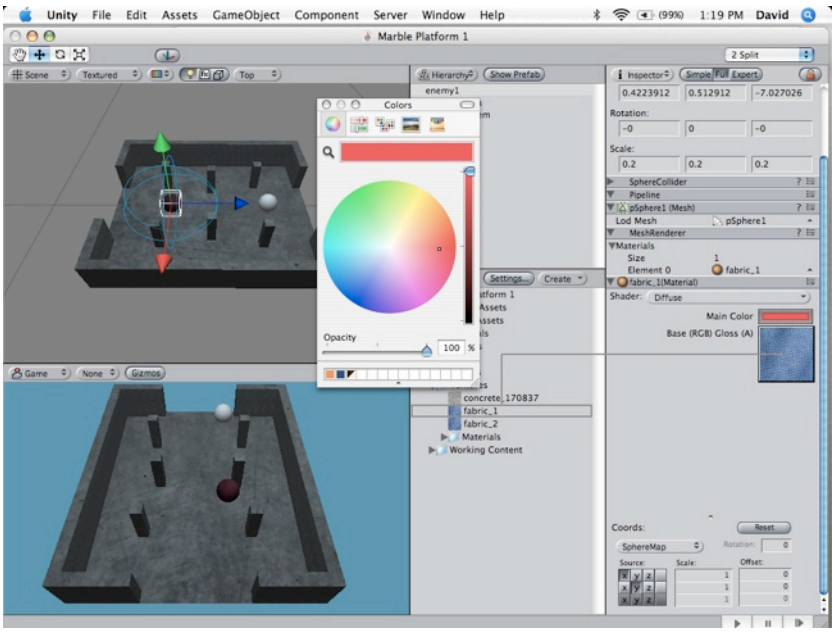
Unity lends itself to both seasoned game developers, and novices who are exploring and learning to the same degree.



As a final step before adding physics and a player control script to our project, we need to add textures the two spheres we created, change the colour of the "enemy" marble, and also duplicate the "enemy" to create a second enemy marble.

Select the "enemy1" sphere we created in the Scene panel. Open your project specific assets folder in the Project list view, and drag the "fabric_1" texture onto the selected "enemy1" sphere in the Scene panel. Make sure the (UV) coordinates are set to normal UV mapping and set the X and Y scales to 0.2.

Click on the Main Color selector and change the sphere's colour to red:



Project Settings

A Very Basic Overview

There are two basic types of project-wide setting to experiment with, and both are most easily found under the Edit menu in the menu bar. These are Project Settings and Render Settings.

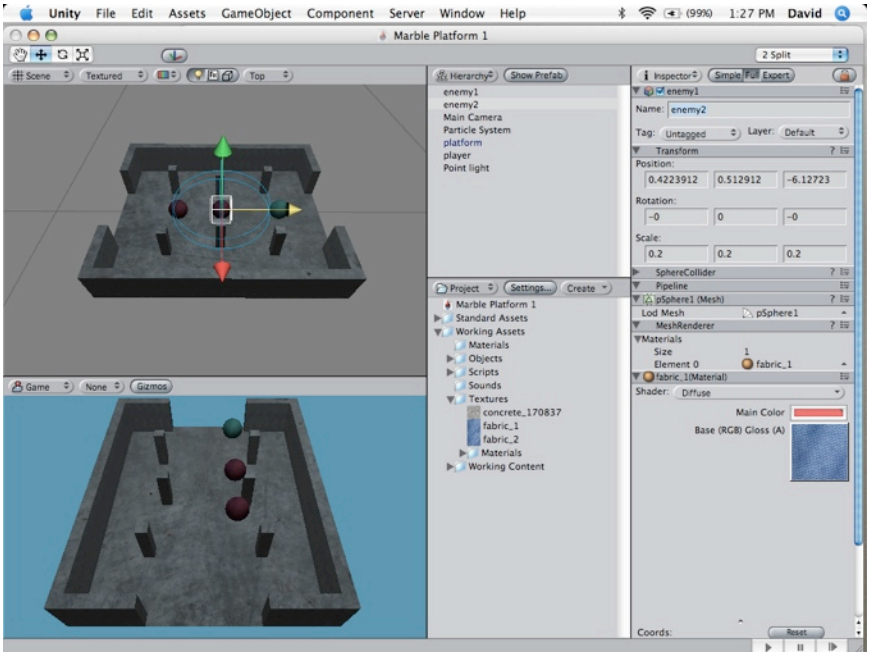
Selecting the Render Settings or one of the many different project Settings (accessible via a submenu) will provide access to those settings' editable attributes in the Inspector panel.

These cover such things as fog, gravity, bounce, physics, input, audio, time, game information, screen resolution, and many other things. Take a few minutes to explore these settings and fiddle with one or two if you feel up to it.

Now do the same thing to the "player" sphere, but use the "fabric_2" asset and change the Main Color attribute of the material to a light green (or whatever other colour you'd like).

To duplicate the "enemy1" sphere select it in the Scene panel and do the usual CMD-D for duplicate and then CMD-V to paste the copy into the scene. It will appear in the same position as the first one so use the move tool to move it in the scene. With the duplicated sphere still selected, use the Inspector panel to rename it "enemy2".

This is what your project will look like at this point.



As a final step here, let's really enhance the look of the spheres by changing one of their settings in the Inspector panel.

Select each sphere in turn and change their Shader setting from *Diffuse* to *Glossy*. This will get show off their specular highlights modulated by the texture's alpha channel. Unity has a great shader collection to explore, by the way.

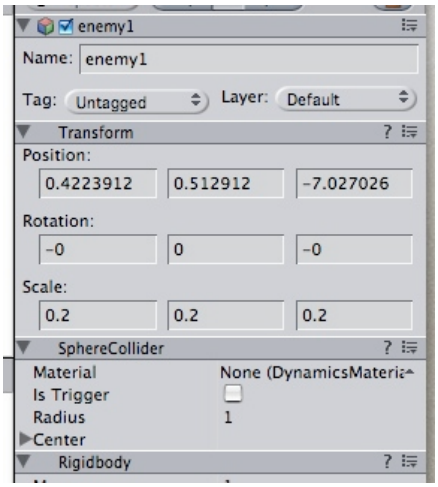
We're done except for looking at and editing/exploring some of the project settings (see *Project Settings* note at left), adding basic physics, and some player control programming that will make our player marble move through our world.

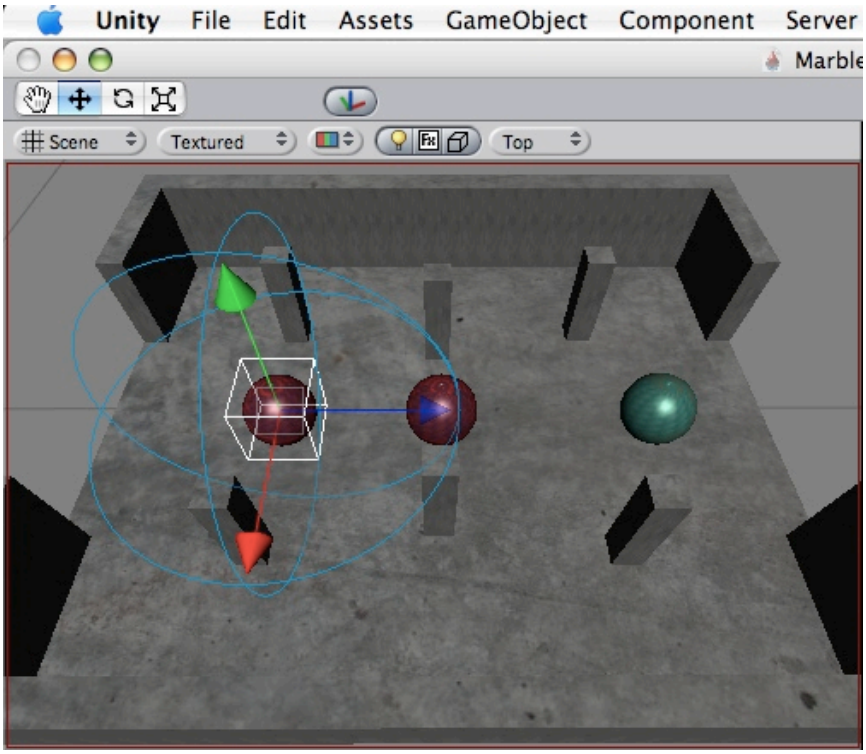
VII – Physics, Collisions and Controls

Adding basic physics, collision detection, and player controls in Unity is actually very simple. We can apply our physics attributes to each individual object, but parent objects (where an object is a template for a number of children objects) can also contain overriding attributes. Let's start by adding a physics component to the platform and three marbles.

Select one of the enemy (red) marbles in the Scene panel and then select from the menu bar Component -> Dynamics -> Rigidbody. This has added a *component* into the Inspector panel with some default values: Mass, Drag, Angular Drag, use Gravity (checked), and Is Kinematic. Leave the default settings for the time being and add the Rigidbody component to the other enemy marble and the player marble as well.

The spheres we created within the Unity environment **already** have another component added automatically by the engine when we created them: SphereCollider. When you click on any of the three spheres in the Scene panel, you'll see a wide, three-axis series of 1 pixel wide bluish circles surrounding them, as shown in this screen shot:





That's the radius of the SphereCollider component (left) and it's too large because when we created the spheres we scaled them down to 0.2. Enter a new value of 0.22 for the Radius and the collision boundary will just slightly exceed the sphere size.

Basic Scripts

Unity Includes Useful Scripts

The Unity engine includes a wide range of standard assets in version 1.0. Included in these are a number of commonly used and useful scripts that can be added as components to game objects.

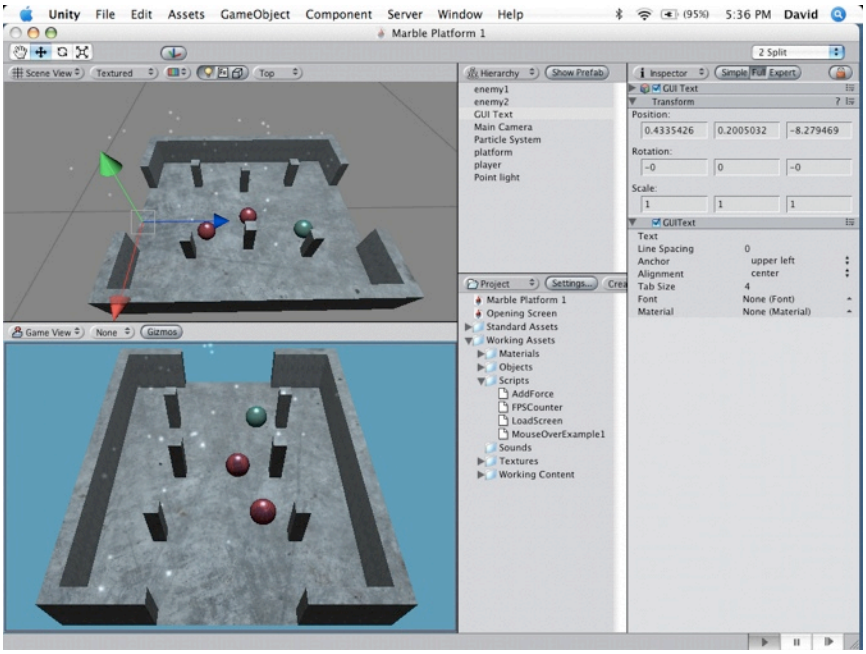
JavaScript is the preferred language to write scripts, but you can also use C# or Boo, a Python-like language developed by Codehaus.

The Unity forums will be a place to discuss, post and download more scripts as the Unity developer community expands. The forums are at http://otee.dk/forums.

Make sure to also reset this SphereCollider Radius value to 0.22 for all three marbles in the game. You could also scale the radius to the sphere size (0.2) quickly by right-clicking the Inspector title bar of the sphere collider component and use the Reset function to reset the sphere to match the collider.

Add a MeshCollider component to the platform object using the previous technique. This creates a collision boundary using the platform's 3D mesh for our marbles to collide with. We do **not** add a Rigidbody component to the platform, however, because we want the platform to remain stationary in space and be unaffected by gravity and mass.

To see our physics in action it's time to *run* the tutorial. Click the arrow Control button in the bottom right of the interface, to run our game. You should see the three marbles fall to the platform and roll slightly on impact. You'll also see the particles start falling too!



Clicking the arrow Play button will reset the objects to their starting positions.

We've now added physics to the game objects; the last thing to do is to add some user controls to the player's marble. We'll do this by adding a script component to the player marble to allow the player to control the marble.

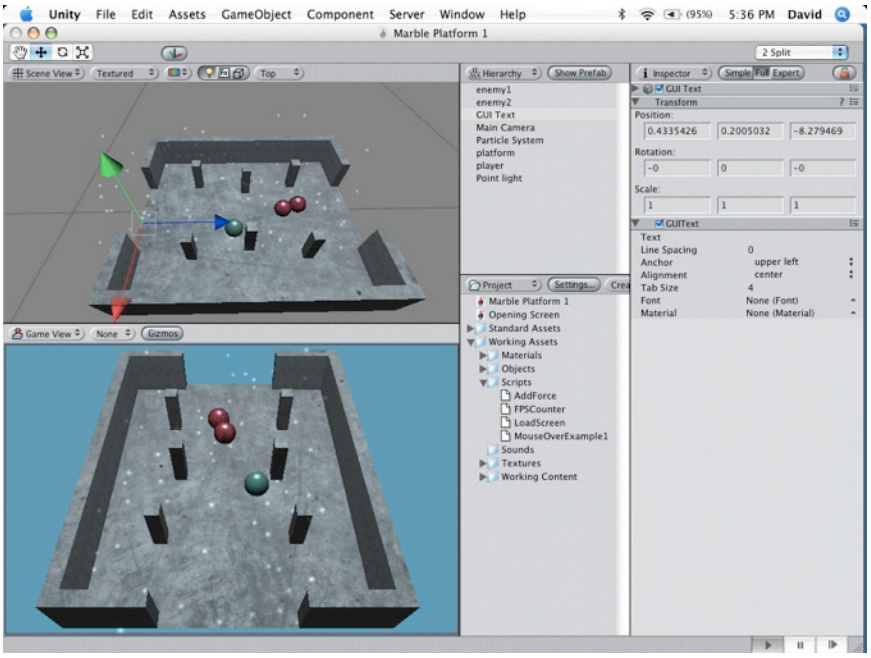
Download the script at http://www.otee.dk/tutorials/BallRolling.js and place it in your Working Assets folder. Make sure it has the .js extension so it's recognized as a Javascript by Unity. Or you could create a new Javascript by using the menu Select Assets -> Create -> Javascript and then pasting the script on page 22 into the Javascript asset you created.

Note: We'll examine the scoring script in detail at the end of Section VIII, on pages 22-23.

After the script has been added we simply select Component -> Scripts -> BallRolling.js while the player marble is selected.

This adds the BallRolling script to the player marble, and allows the player to move the marble using the default ARROW keys.

The following screen shot shows the effects of pushing the two enemy marbles around and off the platform.



That's it! We've now completed a very basic game level that includes objects with realistic physics, collisions, player control, lighting, and particles. Save the project and experiment with some of the settings; e.g., mass of objects, light colours, different particle effects from your spheres, camera positions, etc.

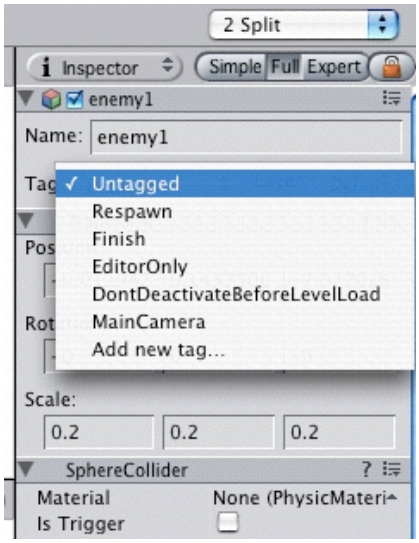
For example, change the enemy spheres' RigidBody component values to be Mass 4, Angular Drag and Drag to 0.1. Set the player sphere Drag to 0.05 and the Angular Drag is 0.1. This gives more realistic motion to the spheres. Also, change the player marble Rolling Marble component to power = 100.

In the last section of the tutorial (IX — Playtime) we'll test and make tweaks to our game, create a banner graphic, and create a Mac standalone application from our project. But before we do that, we have to explain a bit of scripting beyond the simple scripts we added to our tutorial game project in this section.

Tags

A Simple Unity Power Feature

Tags are a simple but powerful way to find and identify objects in the game environment. They can be changed in the Inspector panel in the top pane for objects, as shown below.



VIII – Some Basics About Scripting

Writing scripts for use in your Unity projects allows you to program quickly and easily anything the Unity engine is capable of. Unity's scripting is also accessible and easy to understand. Javascript is the preferred language to write scripts in, but users can also use C# or Boo, a Python-like language developed by Codehaus.

Before writing some basic scripts for our tutorial, I'd suggest looking at some of the Unity example projects and examining the scripts that they contain. This will give a good insight into how the scripts work, how they need to be attached to objects, etc. Also take the time to modify the scripts and then watching what affect your changes have in the game world.

As an example, in the Script Tutorial there is a script called "AddForce" in the Add Force scene attached to the cube called ApplyForceCube. It simply jumps the object when the spacebar is pressed. The script is shown below:

```
var power = 3.0;

function FixedUpdate () {

	// When you press space, we move up the box

	if (Input.GetButton ("Jump")) {

		rigidbody.AddForce (Vector3.up * power);

	}

}
```

Notice that you can add comments your scripts (always a good idea) by starting the comment line with the standard "//". This script can be added to any game object that is a RigidBody.

You can easily modify the amount of "jump force" added to the cube by altering the variable "power". You can also easily modify the Input.GetButton from the standard "jump" (which is the space bar) to another button like "Fire1" (which is CONTROL). Buttons can be modified in the Edit -> Project Settings -> Input in the menu bar.

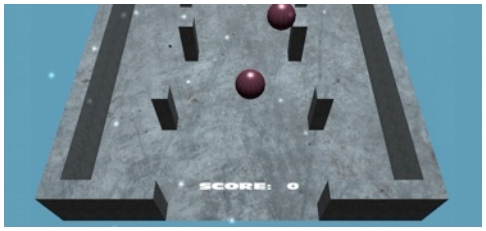
Explore and experiment with the scripts in the other included projects to get a basic understanding of the both syntax and how to effect changes to the projects.

In our tutorial, we can write a couple of simple scripts to add some additional common game functions that many games might need. Specifically, we should write simple scripts that create a rudimentary scoring system, and also a timer counting down that would, in a complete game, create the time a player needed to finish a level. We'll write both scripts using Javascript, and implement both to our scene in a basic way. Feel free to modify and experiment with these scripts too.

Let's start with the scoring system.

Scoring

To implement a basic scoring system, we want our player score to increase each time we knock an enemy marble off the platform to "destroy" it. The simplest way to do that is to have the enemy marble disappear after falling off the platform and down below a certain Y axis value. Then we add one to our player score. There are **two** parts to this: creating a GUIText object to display our score, and writing a script that attaches to our enemy marbles that watches for the Y-axis condition and then adds to the GUIText score.

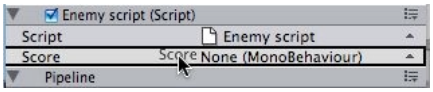


Here is the first script that we need to create. We need an object that will display the score to attach it to. Create a GUIText object and change it's name to "Score". We need a GUIText object so that the information has some place to display on screen. Change it's text attribute to be "Score: 0" and position it on the screen near the bottom opening in the platform walls as shown in the screenshot at left. Then create the following script and save it as "Score.js" in your project's specific assets "Scripts" folder. Finally, attach this script to the GUIText object you made.

This script simply creates a variable called "score" and sets it to be zero. It then implements a function "AddToScore" that changes the GUI text "Score: 0" by adding one to the GUIText display each time an enemy marble falls off the platform.

```
var score = 0;
function AddToScore () {
       ++score;
       guiText.text = "Score: " + score.ToString ();
}
```

Here is the second script that we will attach to **both** of the two enemy marbles. Enter this in a text editor and save it as "Enemy script.js" in your project's specific assets "Scripts" folder.



What is does is declares a variable named "score" which we will assign to the Score script instance we added to the score game object earlier. We assign the score variable by dragging the Score game object onto the score variable in the enemy marble's Inspector panel (shown at left). We do this for both enemies. The script checks to see if the position of the marble is below -2 on the Y-axis and, if it is, will destroy the marble and call the AddToScore function of the Score script.

```
var score : Score;
function Update () {

        if (transform.position.y < -2) {

        Destroy (gameObject);
        score.AddToScore ();

    }

}
```

You can experiment by making a simple change to the score script so that the score increases by 10 each time an enemy falls.

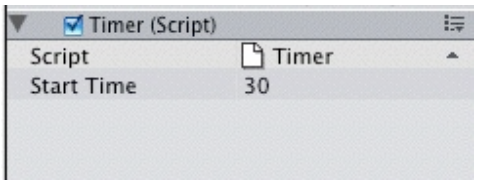
Timer

Let's now create a simple timer. We're going to give the player only 30 seconds to remove the two enemy marbles from the platform. In a complete game, if the player succeeds, we'd show a level winning scene and then advance the player to the next level. If unsuccessful, we might show a screen that allows them to retry the level.

Again, we need a GUIText object to display the time remaining. Create a GUIText object in your scene and rename it "Timer". Change it's text attribute to read 0:30.0. Position it at the top of the platform, as shown in the screen shot at left.

Create the following script, saving it as "Timer.js". Notice the **two** basic functions within the script: one to actually update our GUIText object, and a second longer one to format the timer display correctly.

```
var startTime = 30.0;

function Update () {

	// The time left for player to complete level!
	timeLeft = startTime - Time.time;
	// Don't let the time left go below zero.
	timeLeft = Mathf.Max (0, timeLeft);
	// Format the time nicely
	guiText.text = FormatTime (timeLeft);

}


// Format time like this
// 12[minutes]:34[seconds].5[fraction]

function FormatTime (time)

{

	var intTime : int = time;
	var minutes : int = intTime / 60;
	var seconds : int = intTime % 60;
	var fraction : int = time * 10;
	fraction = fraction % 10;

	// Build string with format
	// 12[minutes]:34[seconds].5[fraction]

	timeText = minutes.ToString () + ":";
	timeText = timeText + seconds.ToString ();
	timeText += "." + fraction.ToString ();
	return timeText;

}
```



Notice anything interesting in the screen shot above? The Timer script area of the GUIText object we made in the Inspector panel actually shows the startTime variable we created.

This is initialized at 30 when we first added the script to the game object.

Note: When you change the script, this value will not update and will remain a 30 second start time. But, if you change it in the Inspector panel, it doesn't update the script, but does become the new start time available.

This is useful for quickly tweaking values without having to go into the script and changing the value there.

Attach this script to the GUIText "Timer" that we made and run the game to see both the scoring and timer functionality!

Some Advanced Scripting – The RollingBall.js Script

The next two pages provide a basic explanation of the BallRolling script we introduced in the previous section on page 17. If you'd prefer to ignore the details of how the script works, feel free to do so. You can begin to learn more about scripting by re-reading this at a later date or by learning about scripting in the second tutorial called Scripting Tutorial.pdf found in Unity's application folder.

So let's take a begining look at the BallRolling.js script and try to figure out what's going on:

```
var power = 50.0;
var drag = 2.0;
private var grounded = false;

function FixedUpdate () {

   // Only apply forces if the ball is on the floor
   if (grounded) {
     // Calculate the direction we want to roll the ball
     // It is relative to the camera
     // * we remove the y component because we want to apply forces only on the 2D plane

     var forward = Camera.main.transform.TransformDirection(Vector3.forward);
     forward.y = 0;
     forward = forward.normalized;

     // Scale the force by the users input and apply it to the rigidbody
     var forwardForce = forward * Input.GetAxis("Vertical") * power;
     rigidbody.AddForce(forwardForce);

     // Calculate the direction in which we want to roll the ball
     // * It is relative to the camera
     // * we remove the y component because we want to apply forces only on the 2D plane

     var right = Camera.main.transform.TransformDirection(Vector3.right);
     right.y = 0;
     right = right.normalized;

     // Scale the force by the users input and apply it to the rigidbody
     var rightForce = right * Input.GetAxis("Horizontal") * power;
     rigidbody.AddForce(rightForce);

     // Apply drag only if the user is not pressing any keys
     if (Mathf.Approximately(Input.GetAxis("Horizontal"),0) &&
     Mathf.Approximately(Input.GetAxis("Vertical"),0))
        rigidbody.drag = drag;
     else
        rigidbody.drag = 0;
   }
   else
        rigidbody.drag = 0;

     // Every frame set grounded to false. OnCollisionStay will enable
     // it again if the rigidbody collides with anything
     grounded = false;
}

function OnCollisionStay() {
     grounded = true;
}
```

1 2 3 4 5 6 7 8 9 10

Yikes. If you're new to programming, you might be asking yourself what exactly is happening in this script? What follows is a basic breakdown of what the script does to the player marble.

1. We set up "variables" called Power, Drag and Grounded. Variables are, at their most basic, things used to store and manipulate data in the form of values (i.e., numeric), that can be modified in the Inspector pane.

Power will be used to determine how much force to apply to the ball, when the key's are pressed. Drag will be used to slow down the ball when no keys are pressed. Grounded is used to determine if the ball is currently touching the ground or not. Only if the ball is grounded will we apply drag and forces to move it around.

2. Next, we've created a series of events inside a "function." A function is something that is defined once, but can be used by a program many times. In some cases, like "Update" functions, they are "called" (run) every frame. Functions are fully explained in the Scripting Tutorial.pdf found in the Unity application folder. FixedUpdate is called every physics frame and allows you to add forces to an object or objects based on user input.

3. We now calculate the forward direction as seen from the camera. We zero out the y component since we want to apply the force only on the horizontal plane. If the camera is looking downwards, we don't want to apply forces downwards on the ball but only forward and to the side. We do this by *normalizing* the forward vector – make it have length = 1. This is necessary because removing the y component of the vector has changed the length but we want to move the ball with the same speed regardless if we are looking mainly downwards, or horizontally, with the camera.

4. We now scale the force we add to the ball (the rigidbody) based on how much input via the keyboard the player has applied.

5-6. These steps are identical to steps 3-4 except that we do it to the right instead of forward and use the "Horizontal" input axis.

7. If there is no input we slow down the ball using drag. Input axes yield values between -1 and 1 where -1 means to the left and 1 means to the right. 0 means neutral (no button pressed). If both horizontal and vertical axis return approximately zero, then we have no player input via the keyboard and we will enable drag.

8. This is simply to test to see if the ball is in the air. If we are in air, we disable drag which makes the physics feel more realistic.

9-10. This is where we will track the ball and determine if it is touching any other colliders. At the end of every frame we set isGrounded to false. OnCollisionStay is called every frame if the sphere touches any other colliders. If it has, we enable isGrounded again. This is repeated every frame so that isGrounded will track if the sphere touches the ground or not during each frame of the game.

This might be a good point to mention that OTEE provides excellent online resources to learn about scripting. They are found at http://www.otee.dk/Documentation/ScriptReference/index.html.

Standalone Notes

User Start Screens and Preferences

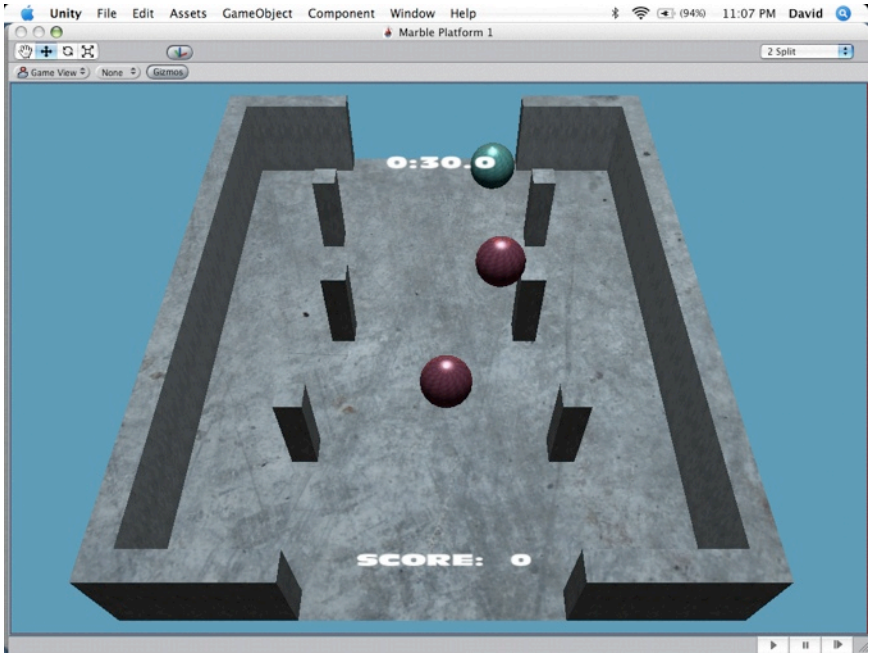
Before you export your project as a finished, standalone application, you have the option of specifying a preferences pane open when the user starts your game program. This is what it looks like (taken from OTEE/Ambrosia's Gooball):

This can be found under Edit -> Project Settings -> Player Settings under the menu bar. The various project settings will appear in the Inspector panel.

You have the option of adding a custom banner image to this dialogue box. The image size needs to be no more than 432 pixels wide by 163 pixels high. It can be made smal ote about textures on page 8. The easiest way to do this is to create a 512 pixel high x 256 pixel wide document in your image editing program, and create your banner image (up to 432x163) in the exact middle of this space.

IX – Playtime

We're going to use this final section of the tutorial to test and make sure our game works as expected, and then make a real standalone application from our project.



Select and expand to full screen by clicking in the Game panel and hitting SPACEBAR and run the game as shown in the screen shot above (the screen shot shows a frame per second counter script added to a GUIText object).

Start your testing by making sure that the marbles fall *onto* the platform (as opposed to *through* the platform). Can you control the green player marble using the ARROW keys? Does the game play and react like you wanted it to? Does the player marble move to quickly or exhibit other odd behaviour (climbing walls, moving far too fast, etc)? Is the frame-per-second counter working?

Some things you may want to adjust include: object mass, drag, acceleration, collision radius, particle effects, camera angle or background colour, etc. Stop the *run* mode and use the Inspector panel to modify each objects attributes as you need. Then retest some more. Review some of the suggested changes we made on page 18.

At this point, we're ready to create our first standalone game application from our project. Create a custom banner image for our game (as explained in *Standalone Notes* at left) and add it to the Edit -> Project Settings -> Player Settings in the Inspector panel, Resolution Dialog Banner, under the menu bar. Also change whatever other attributes of the ProjectSettings you'd like at this point.

Thanks

Thanks To Those Who Helped!

I wasn't really sure I could get this tutorial done. My son, Cal, really wanted to try to make a simple little game, and when Unity came along I jumped at the chance.

Joachim and Nicholas at OTEE were especially helpful. The invited me into the beta test of Unity, and were always quick with answers and always patient with my (usually) very simple questions.

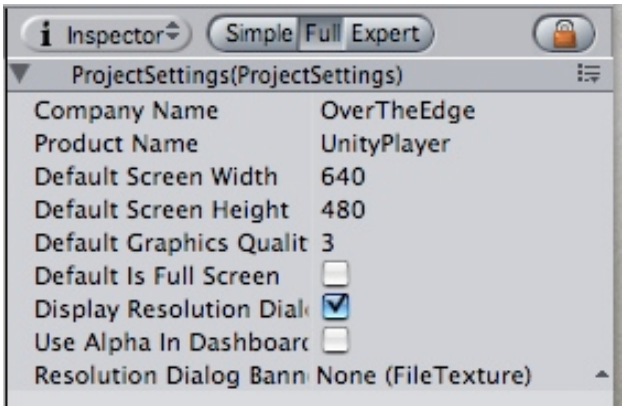
Unity was a wonderful program at version 1, and has improved tremendously as it has matured through version 1.2 and a game developer user base has grown around it. It offers Mac game developers a powerful, easy to use, wonderfully deep game development engine that can save create standalone games for Mac and PC. Best of all, Unity works like a Mac program should, and offers new and seasoned game developers another great Mac application to create cool software for all us gamers.

Comments or questions are always welcome. I can be found around the Unity user forums as DaveyJJ.

This tutorial document will be updated as necessary, and is © 2006 by OTEE and David & Cal Janik-Jones. The project files are available for download at www.otee.dk/tutorials/marble.zip.

Any errors in this tutorial are the authors', and not OTEEs.

24 January 2006. Version 3.



We're done except for exporting a standalone application.

Select File -> Build Game... from the menu bar. Click the top button "Add active scene" to add our current scene to the standalone. Check Build MacOS X Player from the Build Target type and then adjust any other of the optional variables as you need. Click the "Build" button, specify where to save the standalone to, and you're done!

Quit Unity and double-click the application you just built! Here's the dialog box from my tutorial as the game starts: