CMSC 425: Lecture 3
Introduction to Unity

Reading: For further information about Unity, see the online documentation, which can be found at `http://docs.unity3d.com/Manual/`. The material on Unity scripts is largely based on lecture notes by Diego Perez from the University of Essex, `http://orb.essex.ac.uk/ce/ce318/`. Note that we will only have time to cover part of this in class, but please read the *entire lecture*. Also, visit the many good tutorials provided by Unity on the Internet.

Unity: Unity3D is a widely-used cross-platform game develop system. It consists of a game engine and an integrated development environment (IDE). It can be used to develop games for many different platforms, including PCs, consoles, mobile devices, and deployment on the Web. In this lecture, we will present the basic elements of Unity. However, this is a complex system, and we will not have time to delve into its many features. A good starting point for learning about Unity is to try the many tutorials available on the Unity Tutorial Web site.

Unity Basic Concepts: The fundamental structures that make up Unity are the same as in most game engines. As with any system, there are elements and organizational features that are unique to this particular system.

Project: The *project* contains all the elements that makes up the game, including models, assets, scripts, scenes, and so on. Projects are organized hierarchically in the same manner as a file-system's folder structure.

Scenes: A *scene* contains a collection of game objects that constitute the world that the player sees at any time. A game generally will contain many scenes. For example, different levels of a game would be stored as different scenes. Also, special screens (e.g., an introductory screen), would be modeled as scenes that essentially have only a two-dimensional content.

Packages: A *package* is an aggregation of game objects and their associated meta-data. Think of a package in the same way as library packages in Java. They are related objects (models, scripts, materials, etc.). Here are some examples:

- a collection of shaders for rendering water effects
- particle systems for creating explosions
- models of race cars for a racing game
- models of trees and bushes to create a woodland scene

Unity provides a number standard packages for free, and when a new project is created, you can select the packages that you would like to have imported into your project.

Prefabs: A *prefab* is a template for grouping various assets under a single header. Prefabs are used for creating multiple instances of a common object. Prefabs are used in two common ways. First, in designing a level for your game you may have a large number of copies of a single element (e.g., street lights). Once designed, a street light prefab can be instantiated and placed in various parts of the scene. If you decide to want to change the intensity of light for all the street lights, you can modify the prefab, and this will cause all the instances to change. A second use is to generate dynamic game objects.

For example, you could model an explosive shell shot from a cannon as a prefab. Each time the cannon is shot a new prefab shell would be instantiated (through one of your scripts). In this way each new instance of the shell will inherit all the prefabs properties, but it will have its own location and state.

Game Objects: The *game objects* are all the "things" that constitute your scene. Game objects not only include concrete objects (e.g., a chair in a room), but also other elements that reside in space such as light sources, audio sources, and cameras. Empty game objects are very useful, since they can to serve as parent nodes in the hierarchy. Every game object (even empty ones) has a position and orientation space. This, it can be moved, rotated and scaled. (As mentioned above, whenever a transformation is applied to a parent object, it is automatically propagated to all of this object's descendants descendants.)

Game objects can be used for invisible entities that are used to control a game's behavior. (For example, suppose that you want a script to be activated whenever the player enters a room. You could create an invisible portal object covering the door to the room that triggers an event whenever the player passes through it.) Game objects can be enabled or disabled. (Disabled objects behave as if they do not exist.) It is possible to associate various elements with game objects, called components, which are described below.

Components: As mentioned above, each game object is defined by a collection of associated elements. These are called *components*. The set of components that are associated with a game object depend on the nature of object. For example, a light source object is associated with color and intensity of the light source. A camera object is associated with various properties of how the projection is computed (wide-angle or telephoto). Physical objects of the scene are associated with many different components. For example, these might include:

- A *mesh filter* and *mesh renderer* are components that define the geometric surface model for the object and the manner in which it is drawn, respectively.
- A *rigid body* component that specifies how the object moves physically in space by defining elements like the object's mass, drag (air resistance), and whether the object is affected by gravity.
- A *collider* which is an imaginary volume that encloses the object and is used to determine whether the object collides with other objects from the scene. (In theory, the object's mesh describes it shape and hence be used for computing collisions, but for the sake of efficiency, it is common to use a much simpler approximating shape, such as a bounding box or a bounding sphere, when detecting collisions.)
- Various *surface materials*, which describe the object's color, texture, and shading.
- Various *scripts*, which control how the object behaves and how it reacts to its environment. One example of a script is a *player controller*, which is used for the player object and describes how the object responds to user inputs. (See below for more information.)

The various components that are associated with an game object can be viewed and edited in the Inspector window (described below).

Assets: An *asset* is any resource that will be used as part of an object's component. Examples include meshes (for defining the shapes of objects), materials (for defining shapes),

physics materials (for defining physical properties like friction), and scripts (for defining behaviors).

Scripts: A *script* is a chunk of code that defines the behavior of game objects. Scripts are associated with game objects. There are various types of scripts classes, depending on the type of behavior being controlled. Because interactive game programming is *event-driven*, a typical script is composed as a collection of functions, each of which is invoked in response to a particular event. (E.g., A function may be invoked when this object collides with another object.) Typically, each of these functions performs some simple action (e.g., moving the game object, creating/destroying game objects, triggering events for other game objects), and then returns control to the system.

Overview of the Unity IDE: Having described the basic concepts underlying Unity, let us next take a quick look at the Unity user interface. As mentioned above, Unity provides a integrated development environment in which to edit and develop your game. While the user interface is highly configurable, there are a few basic windows which are the most frequently used (see Fig. 1).

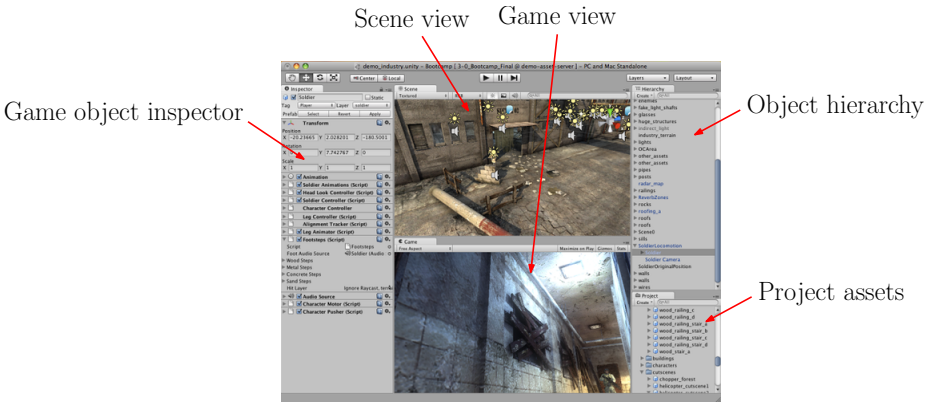

Fig. 1: Unity IDE.

Scene View: This window shows all the elements of the current scene. (See description below for what a scene is.) Most editing of the scene is done through the scene view, because it provides access to low-level and hidden aspects of the objects. For example, this view will show you where the camera and light sources are located. In contrast, the Game View, shows the game as it would appear to the player.

Game View: This window shows the elements of the scene as they would appear to the player.

Inspector: At any time there is an *active* game object (which the designer selects by clicking on the object or on its entry in the hierarchy). This window provides all the component information associated with this object. At a minimum, this includes its *transform* indicating its position and orientation in space. However it also has entries for each of the components (mesh renderer, rigid body, collider, etc.) associated with this object.

Hierarchy: This window shows all the game objects that constitute the current scene. (Scenes are discussed below). As its name suggests, game objects are stored hierar-

chically in a tree structure. This makes it possible so that transformations applied to a parent object are then propagated to all of its descendents. For example, if a building is organized as a collection of rooms (descended from the building), and each room is organized as a collection of pieces of furniture (descended from the room), then moving the building will result in all the rooms and pieces of furniture moving along with it.

Project: The project window contains all of the assets that are available for you to use. Typically, these are organized into folders, for example, according to the asset type (models, materials, audio, prefabs, scripts, etc.).

Scripting in Unity: As mentioned above, scripting is used to describe how game objects behave in response to various events, and therefore it is an essential part of the design of any game. Unity supports two different scripting languages: C# and UnityScript, a variant of JavaScript. (C# is the better supported. At a high level, C# is quite similar to Java, but there are minor variations in syntax and semantics.) Recall that a script is an example of a component that is associated with an game object. In general, a game object may be associated with multiple scripts.

Geometric Elements: Unity supports a number of objects to assist with geometric processing. We will discuss these objects in greater detail in a later lecture, but here are a few basic facts.

Vector3: This is standard (x, y, z) vector. As with all C# objects, you need to invoke "new" when creating a Vector3 object. The following generates a Vector3 variable u with coordinates $(1, 2, -3)$:

```
Vector3 u = new Vector3(1, 2, -3);
```

The orientation of the axes follows Unity's (mathematically nonstandard) convention that the y-axis is directed upwards, the x-axis points to the viewer's right, and the z-axis points to the viewer's forward direction. (Of course, as soon as the camera is repositioned, these orientations change.)

It is noteworthy that Unity's axis forms what is called a *left-handed coordinate system*, which means that $x \times y = -z$ (as opposed to $x \times y = z$, which holds in most mathematics textbooks as well as other 3D programming systems, such as UE4 and Blender).

To make it a bit more natural in programming, Unity provides function calls that return the unit vectors in natural directions. For example, Vector3.right return $(1, 0, 0)$, Vector3.up returns $(0, 1, 0)$, and Vector3.forward returns $(0, 0, 1)$. Others include left, down, back, and zero.

Ray: Rays are often used in geometric programming to determine the object that lies in a given direction from a given location. (Think of shooting a laser beam from a point in a direction and determining what it hits.) A ray is specified by giving its origin and direction. The following creates a ray starting at the origin and directed along the x-axis.

```
Ray ray = new Ray(Vector3.zero, Vector3.right);
```

We will discuss how to perform ray-casting queries in Unity and how these can be applied in your programs.

Quaternion: A quaternion is a structure that represents a rotation in 3-dimensional space. There are many ways to provide Unity with a rotation. The two most common are

through the use of *Euler angles*, which means specifying three rotation angles, one about the x-axis, one about the y-axis, and one about the z-axis. The other is by specifying a Vector3 as an axis of rotation and a rotation angle about this axis. For example, the following both create the same quaternion, which performs a 30° rotation about the vertical (that is, y) axis.

```
Quaternion q1 = Quaternion.Euler(0, 30, 0);
Quaternion q2 = Quaternion.AngleAxis(30, Vector3.up);
```

Transform: Every game object in Unity is associated with an object called its *transform*. This object stores the position, rotation, and scale of the object. You can use the transform object to query the object's current position (transform.position) and rotation (transform.eulerAngles).

You can also modify the transform to reposition the object in space. These are usually done indirectly through functions that translate (move) or rotate the object in space. Here are examples:

```
transform.Translate(new Vector3(0, 1, 0)); // move up one unit
transform.Rotate(0, 30, 0); // rotate 30 degrees about y-axis
```

You might wonder whether these operations are performed relative to the global coordinate system or the object's local coordinate system. The answer is that there is an optional parameter (not shown above) that allows you to select the coordinate system about which the operation is to be interpreted.

Recall that game objects in Unity reside within a hierarchy, or tree structure. Transformations applied to an object apply automatically to all the descendants of this object as well. The tree structure is accessible through the transform. For example, transform.parent returns the transform of the parent, and transform.parent.gameObject returns the Unity game object associated with the parent. You can set a transforms parent using transform.SetParent(t), where t is the transform of the parent object. It is also possible to enumerate the children and all the descendants of a transform.

Structure of a Typical Script: A game object may be associated with a number of scripts. Ideally, each script is responsible for a particular aspect of the game object's behavior. The basic template for a Unity script, called MainPlayer, is given in the following code block.

Script template

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {
    void Start () {
        // ... insert initialization code here
    }
    void Update () {
        // ... insert code to be repeated every update cycle
    }
}
```

Observe a few things about this code fragment. First, the first using statements provides access to class objects defined for the Unity engine, and the second provides access to built-in collection data structures (ArrayList, Stack, Queue, HashTable, etc.) that are part of C#. The main class is MainPlayer, and it is a subclass of MonoBehaviour. All script objects in Unity are subclasses of MonoBehaviour.

Many script class objects will involve: (1) some sort of initialization and (2) some sort of incremental updating just prior to each refresh cycle (when the next image is drawn to the display). The template facilitates this by providing you with two blank functions, Start and Update, for you to fill in. Of course, there is no requirement to do so. For example, your script may require no explicit initializations (and thus there is no need for Start), or rather than being updated with each refresh cycle, your script may be updated in response to specific events such as collisions (and so there would be no need for Update).

Awake versus Start: There are two Unity functions for running initializations for your game objects, Start (as shown above) and Awake. Both functions are called at most once for any game object. Awake will be called first, and is called as soon as the object has been initialized by Unity. However, the object might not yet appear in your scene because it has not been *enabled*. As soon as your object is enabled, the Start is called.

Let us digress a moment to discuss enabling/disabling objects. Unity game objects can be "turned-on" or "turned-off" in two different ways (without actually deleting them). In particular, objects can be *enabled* or *disabled*, and objects can be *active* or *inactive*. (Each game object in Unity has two fields, enabled and active, which can be set to true or false.) The difference is that disabling an object stops it from being rendered or updated, but it does not disable other components, such as colliders. In contrast, making an object *inactive* stops all its components.

For example, suppose that some character in your game is spawned only after a particular event takes place (you cast a magic spell). The object can initially be declared to be *disabled*, and later when the spawn event occurs, you ask Unity to enable it. Awake will be called on this object as soon as the game starts. Start will be called as soon as the object is enabled. If you later disable and object and re-enable it, Start *will not* be called again. (Both functions are called at most once.)

To make your life simple, it is almost always adequate to use just the Start function for one-time initializations. If there are any initializations that *must* be performed just as the game is starting, then Awake is one to use.

Controlling Animation Times: As mentioned above, the Update function is called with each update-cycle of your game. This typically means that every time your scene is redrawn, this function is called. Redraw functions usually happen very quickly (e.g., 30–100 times per second), but they can happen more slowly if the scene is very complicated. The problem that this causes is that it affects the speed that objects appear to move. For example, suppose that you have a collection of objects that spin around with constant speed. With each call to Update, you rotate them by 3°. Now, if Update is called twice as frequently on one platform than another, these objects will appear to spin twice as fast. This is not good!

The fix is to determine how much time as elapsed since the last call to Update, and then scale the rotation amount by the elapsed time. Unity has a predefined variable, Time.deltaTime,

that stores the amount of elapsed time (in seconds) since the last call to Update. Suppose that we wanted to rotate an object at a rate of 45° per second about the vertical axis. The Unity function transform.Rotate will do this for us. We provide it with a vector about which to rotate, which will usually be $(0, 1, 0)$ for the up-direction, and we multiply this vector times the number of degrees we wish to have in the rotation. In order to achieve a rotation of 45° per second, we would take the vector $(0, 45, 0)$ and scale it by Time.deltaTime in our Update function. For example:

Constant-time rotation

```
void Update () {
    transform.Rotate (new Vector3 (0, 45, 0) * Time.deltaTime);
}
```

By the way, it is a bit of a strain on the reader to remember which axis points in which direction. The Vector3 class defines functions for accessing important vectors. The call Vector3.up returns the vector $(0, 1, 0)$. So, the above call would be equivalent to transform.Rotate (Vector3.up * 45 * Time.deltaTime).

Update versus FixedUpdate: While we are discussing timing, there is another issue to consider. Sometimes you want the timing between update calls to be predictable. This is true, for example, when updating the physics in your game. If acceleration is changing due to gravity, you would like the effect to be applied at regular intervals. The Update function does not guarantee this. It is called at the refresh rate for your graphics system, which could be very high on a high-end graphics platform and much lower for a mobile device.

Unity provides a function that is called in a predictable manner, called FixedUpdate. When dealing with the physics system (e.g., applying forces to objects) it is better to use FixedUpdate than Update. When using FixedUpdate, the corresponding elapsed-time variable is Time.fixedDeltaTime. (I've read that Time.fixedDeltaTime is 0.02 seconds, but I wouldn't bank on that.)

While I am discussing update functions, let me mention one more. LateUpdate() is called after all Update functions have been called but before redrawing the scene. This is useful to order script execution. For example a follow-camera should always be updated in LateUpdate because it tracks objects that might have moved due to other Update function calls.

Is this confusing? Yes! If you are unsure, it is "usually" safe to put initialization code in Start and update code in Update. If the behavior is not as expected, then explore these other functions. By the way, it gets much more complicated that this. If you really want to be scared, check out the *Execution Order of Event Functions* in the Unity manual for the full documentation.

Accessing Components: As mentioned earlier, each game object is associated with a number of defining entities called its *components*. The most basic is the transform component, which describes where the object is located. Most components have constant values, and can be set in the Unity editor (for example, by using the AddComponent command. However, it is often desirable to modify the values of components at run time. For example, you can alter the

buoyancy of a balloon as it loses air or change the color of a object to indicate the presence of damage.

Unity defines class types for each of the possible components, and you can access and modify this information from within a script. First, in order to obtain a reference to a component, you use the command GetComponent. For example, to access the rigid-body component of the game object associated with this script, you could invoke. Recall that this component controls the physics properties of the object.

```
Rigidbody rb = GetComponent<Rigidbody>(); // get rigidbody component
```

This returns a reference rb to this object's rigid body component, and similar calls can be made to access any of the other components associated with a game object. (By the way, this call was not really needed. Because the rigid body is such a common component to access, every MonoBehaviour object has a member called rigidbody, which contains a reference to the object's rigid body component, or null if there is none.)

Public Variables and the Unity Editor: One of the nice features that the Unity IDE provides is the ability to modify the member variables of your game objects directly within the editor, even as your game is running. For example, suppose that you have a moving object that has an adjustable parameter. Consider the following code fragment that is associated with a floating ball game object. The script defines a public member variable floatSpeed to control the speed at which a ball floats upwards.

```
public class BallBehavior : MonoBehaviour {

  public float floatSpeed = 10.0f; // how fast ball floats up
  public float jumpForce = 4.0f;   // force applied when ball jumps
  ...
}
```

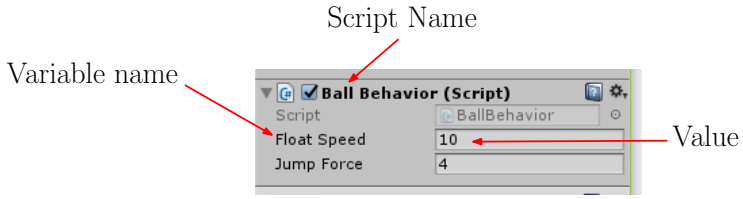

Fig. 2: Editing a public variable in the inspector.

When you are running the program in the Unity editor, you can adjust the values of floatSpeed and jumpForce until you achieve the desired results. (Note that if you modify the variable while the game is running, it will be reset to its original value when the game terminates.) If you like, you can then fix their values in the script and make them private.

Note that there are three different ways that the member variable floatSpeed could be set:

(1) It could be initialized as part of its declaration, public floatSpeed = 6.0f;

(2) It could be set by you in the Unity editor (as above)

(3) It could be initialized in the script, e.g., in the Start() function.

Note that (3) takes precedence over (2), which takes precedence over (1).

By the way, you can do this not only for simple variables as above, but you can also use this mechanism for passing game objects through the public variables of a script. Just make the game object variable public, then drag the game object from the hierarchy over the variable value in the editor.

Object References by Name or Tag: Since we are on the subject of how to obtain a reference from one game object to another, let us describe another mechanism for doing this. Each game object is associated with a *name*, and its can be also be associated with one more *tags*. Think of a name as a unique identifier for the game object (although, I don't think there is any requirement that this be so), whereas a tag describes a type, which might be shared by many objects.

Both names and tags are just a string that can be associated with any object. Unity defines a number of standard tags, and you can create new tags of your own. When you create a new game object you can specify its name. In each object's inspector window, there is a pull-down menu that allows you associate any number of tags with this object.

Here is an example of how to obtain a the main camera reference by its name:

```
GameObject camera = GameObject.Find ("Main Camera");
```

Suppose that we assign the tag "Player" with the player object and "Enemy" with the various enemy objects. We could access the object(s) through the tag using the following commands:

```
GameObject player = GameObject.FindWithTag("Player");
GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy");
```

In the former case, we assume that there is just one object with the given tag. (If there is none, then null is returned. If there is more than one, it returns one one of them.) In the latter case, all objects having the given tag are returned in the form of an array.

Note that there are many other commands available for accessing objects and their components, by name, by tag, or by relationship within the scene graph (parent or child). See the Unity documentation for more information. The Unity documentation warns that these access commands can be relatively *slow*, and it is recommended that you execute them once in your Start() or Awake() functions and save the results, rather than invoking them with every update cycle.

Accessing Members of Other Scripts: Often, game objects need to access member variables or functions in other game objects. For example, your enemy game object may need to access the player's transform to determine where the player is located. It may also access other functions associated with the player. For example, when the enemy attacks the player, it needs to invoke a method in the player's script that decreases the player's health status.

```
public class PlayerController : MonoBehaviour {
  public void DecreaseHealth() { ... } // decrease player's health
}
```

```
public class EnemyController : MonoBehaviour {
  public GameObject player; // the player object
  void Start () {
    GameObject player = GameObject.Find("Player");
  }
  void Attack () {    // inflict health loss on player
    player.GetComponent<PlayerController>().DecreaseHealth();
  }
}
```

Note that we placed the call to GameObject.Find in the Start function. This is because this operation is fairly slow, and ideally should be done sparingly.

Colliders and Triggers: Games are naturally *event driven*. Some events are generated by the user (e.g., input), some occur at regular time intervals (e.g., Update()), and finally others are generated within the game itself. An important class of the latter variety are collision events. Collsions are detected by a component called a *collider*. Recall that this is a shape that (approximately) encloses a given game object.

Colliders come in two different types, *colliders* and *triggers*. Think of colliders as solid physical objects that should not overlap, whereas a *trigger* is an invisible barrier that sends a signal when crossed.

For example, when a rolling ball hits a wall, this is a collider type event, since the ball should not be allowed to pass through the wall. On the other hand, if we want to detect when a player enters a room, we can place an (invisible) trigger over the door. When the player passes through the door, the trigger event will be generated.

There are various event functions for detecting when an object enters, stays within, or exits, collider/trigger region. These include, for example:

- For colliders: void OnCollisionEnter(), void OnCollisionStay(), void OnCollisionExit()
- For triggers: void OnTriggerEnter(), void OnTriggerStay(), void OnTriggerExit()

More about RigidBody: Earlier we introduced the rigid-body component. What can we do with this component? Let's take a short digression to discuss some aspects of rigid bodies in Unity. We can use this reference to alter the data members of the component, for example, the body's mass:

```
rb.mass = 10f;     // change this body's mass
```

Unity objects can be controlled by physics forces (which causes them to move) or are controlled directly by the user. One advantage of using physics to control an object is that it will automatically avoid collisions with other objects. In order to move a body that is controlled by physics, you do not set its velocity. Rather, you apply forces to it, and these forces affect it velocity. Recall that from physics, a force is a vector quantity, where the direction of the vector indicates the direction in which the force is applied.

```
rb.AddForce(Vector3.up * 10f);     // apply an upward force
```

Sometimes it is desirable to take over control of the body's movement yourself. To turn off the effect of physics, set the rigid body's type to *kinematic*.

```
rb.isKinematic = true;  // direct motion control---bypass physics
```

Once a body is kinematic, you can directly set the body's velocity directly, for example.

```
rb.velocity = Vector3(0f, 10f, 0f);  // move up 10 units/second
```

(If the body had not been kinematic, Unity would issue an error message if you attempted to set its velocity in this way.) By the way, since the y-axis points up, the above statement is equivalent to setting the velocity to Vector3.up * 10f. This latter form is, I think, more intuitive, but I just wanted to show that these things can be done in various ways.

Kinematic and Static: In general, Physics computations can be expensive, and Unity has a number of tricks for optimizing the process. As mentioned earlier, an object can be set to *kinematic*, which means that your scripts control the motion directly, rather than physics forces. Note that this only affects motion. Kinematic objects still generate events in the event of collisions and triggers.

Another optimization involves static objects. Because many objects in a scene (such as buildings) never move, you can declare such objects to be *static*. (This is indicated by a check box in the upper right corner of the object's Inspector window.) Unity does not perform collision detection between two static objects. (It checks for collisions/triggers between static-to-dynamic and dynamic-to-dynamic, but not static-to-static.) This can save considerable computation time since many scenes involve relatively few moving objects. Note that you can alter the static-dynamic property of an object, but the documentation warns that this is somewhat risky, since the physics engine precomputes and caches information for static objects, and this information might be erroneous if an object changes this property.

Event Functions: Because script programming is *event-driven*, most of the methods that make up MonoBehaviour scripts are event callbacks, that is, functions that are invoked when a particular event occurs. Examples of events include (1) initialization, (2) physics events, such as collisions, and (3) user-input events, such as mouse or keyboard inputs.

Unity passes the control to each script intermittently by calling a determined set of functions, called *Event Functions*. The list of available functions is very large, here are the most commonly used ones:

Initialization: Awake and Start as mentioned above.

Regular Update Events: Update functions are called regularly throughout the course of the game. These include redraw events (Update and LateUpdate) and physics (or any other regular time events) (FixedUpdate).

GUI Events: These events are generated in response to user-inputs regarding the GUI elements of your game. For example, if you have a GUI element like a push-down button, you can be informed when a mouse button has been pressed down, up, or is hovering over this element with callbacks such as void OnMouseDown(), void OnMouseUp(), void OnMouseOver().They are usually processed in Update or FixedUpdate.

Physics Events: These include the collider and trigger functions mentioned earlier (such as OnCollisionEnter, OnTriggerEnter).

There are *many* things that we have not listed. For further information, see the Unity user manual.

Loading/Instantiating Prefabs: As mentioned above, a *prefab* is a game object that can be generated, or *instantiated*, at run time. Usually, prefabs are stored among your assets within a folder called "Resources". (This appears at the top level of your Project-view window in the Unity editor, or equivalently, in your Assets folder for your project.) This is a two-step process. First, the prefab is loaded from disk and stored internally as a game object. Next, each time you need a new copy of this object, you instantiate the game object.

For example, suppose that you have written a game where a rocket ship can shoot missiles. In the Unity editor, you create a game object called Missile and drag it from the hierarchy into the Project window within the directory *Resources*. (It will appear on disk as Assets/Resources/Missile.prefab.) Next, suppose that the command for shooting the missile is handled within your script for controlling the rocket ship, say, RocketShipController. In order to use the missile object, you must first load the prefab. Since loading takes time (to copy from the disk) you would do this once when your rocket-ship controller starts.

```
public class RocketShipController : MonoBehaviour {

  public GameObject mPrefab;

  void Start () {
    GameObject mPrefab = Resources.Load("Missile") as GameObject;
  }
```

Now that our missile has been loaded, we can instantiate it as needed. Let us suppose that we have a function Shoot that launches the missile from the rocket ship's current location (transform.position) oriented in the same direction that the rocket ship is facing (transform.rotation). In order to send it on its way, we should give it some positive velocity. To do this, we take a scaled copy of the forward vector associated with the rocket-ship's transform. This can be acquired with the call transform.TransformDirection(Vector3.forward * 10). Intuitively this means, "take the forward vector for the rocket ship, scale by a factor of 10, and then transform this local vector into the world-coordinate system."

```
  // ... (Later within RocketShipController)

  void ShootMissile () {
    GameObject m = Instantiate(mPrefab, transform.position,
                                        transform.rotation);
    m.velocity = transform.TransformDirection(Vector3.forward*10);
  }
```

Now, whenever we invoke this function we will get a new copy of the Missile game object.

Coroutines: (Optional material)

Anyone who has worked with event-driven programming for some time has faced the frustration that, while we programmers like to think *iteratively*, in terms of loops, our event-driven

graphics programs are required to work *incrementally*. The event-driven structure in an intrinsic part of interactive computer graphics has the same general form: wake up (e.g., at every refresh cycle), make a small incremental change to the state, and go back to sleep (and let the graphics system draw the new scene). In order to make our iterative programs fit within the style, we need to "unravel" our loops to fit this awkward structure.

Unity has an interesting approach to helping with this issue. Through a language mechanism, called *coroutines*, it is possible to implement code that *appears* to be iterative, and yet behaves *incrementally*. When you call a function, it runs to completion before returning. This effectively means that the function cannot run a complete loop, e.g., for some animation, that runs over many refresh cycles. As an example, consider the following code snippet that gradually makes a object transparent until it becomes invisible. Graphics objects are colored using a 4-element vector, called RGBA. The R, G, and B components describe the red, green, and blue components (where 1 denotes the brightest value and 0 the darkest). The A component is called the colors *alpha* value, which encodes its level of opacity (where 1 is fully opaque and 0 is fully transparent).

```
void Fade() {  // gradually fade from opaque to transparent
  for (float f = 1f; f >= 0; f -= 0.1f) {
    Color c = renderer.material.color;
    c.a = f;
    renderer.material.color = c;
    // ... we want to redraw the object here
  }
}
```

Unfortunately, this code does not work, because we cannot just interrupt the loop in the middle to redraw the new scene. A *coroutine* is like a function that has the ability to pause execution and return (to Unity) but then to continue where it left off on the following frame. To do this, we use the yield return construct from C#, which allows us to call a function multiple times, but each time it is called, it starts not from the beginning, but from where it left off. Such a function has a return type of an iterator, but we will not worry about this for this example.

```
IEnumerator Fade() { // gradually fade from opaque to transparent
  for (float f = 1f; f >= 0; f -= 0.1f) {
    Color c = renderer.material.color;
    c.a = f;
    renderer.material.color = c;
    yield return null; // return to Unity to redraw the scene
  }
}
```

The next time this function is called, it resumes just *after* the return statement in order to start the next iteration of the loop. (Pretty cool!)

If you want to control the timing, so that this happens, say, once every tenth of a second, you can add a delay into the return statement, "yield return new WaitForSeconds(0.1f)".