UNIVERSITY OF ST. THOMAS

Unity Game Development

CISC 243 Independent Study

by

Miguel A. Velez

Faculty Advisor: Dr. Patrick L. Jarvis

Summer 2014

Table of Contents

Chapter 1

Introduction

1.1 Installation

- Go to `http://unity3d.com/unity/download`.
- Download the latest free version, otherwise, you can rent it for $75/month or buy it for $1500 as of the end of May 2014.
- Open the `.exe` file and follow the instructions.
- Create a Unity Account at `http://forum.unity3d.com/` to access tutorials, answers, and feedback from the Unity community.

1.2 Creating a project

- Open Unity.
- if you do not have any projects, you will be prompted with a `Project Wizard` window.
- Select the project location.
- Select what packages you want to import. To start, do not import any packages.
- Select if you want to do a 2D or 3D project.
- Click `Create`.
- If you want to create a project within Unity, click `File`> `New Project`.
- Follow the steps above to create a project.

1.3 Importing a project

- Inside Unity, go to `Window`> `Asset Store`.
- Browse the multiple finished projects.
- Click the `Download` button to download the entire project.
- After the download is done, an `Importing Package` window appears.
- Click the `Import` button to save the project in your computer.

1.4 Creating a scene

- Inside Unity, go to `File`> `New Scene`.
- If you do not see you scene, go to the top right corner and change the layout to `Default`.

1.5 Navigation within a scene

- `Alt + left click` = move around a point in space.
- `Alt + middle click` = move around scene.
- `Alt + right click` = zoom in and out.
- `F` = focuses on an selected object.
- `Right click + w` = move forward.
- `Right click + s` = move backward.
- `Right click + d` = move right.
- `Right click + a` = move left.
- `Right click + e` = move up.
- `Right click + q` = move down.

1.6 Creating with primitives

- These are objects that can be created directly within Unity.
- Cube, Sphere, Capsule, Cylinder, Plane, and Quad.
- These can be added to a scene via `GameObject> Create Other`.

1.7 Manipulate objects

- On the top left corner there are four buttons to manipulate objects.
- The hand allows you to select an object. The keyboard shortcut is `Q`.
- The up, down, left, and right arrows allow you to move the object in the `X`, `Y`, and `Z` axis. The keyboard shortcut is `W`.
- To move the object in a specific direction, click on one of the arrows that appears on the object and drag it to your desired position.
- If you click and drag from the middle of the object, it moves in all directions at once.
- The curved arrows allow to rotate the object. The keyboard shortcut is `E`.
- To rotate the object in a specific direction, click on one of the lines that appear on the object and drag to rotate.
- If you click and drag on the middle of the object, it rotates in all directions at once.
- The diagonal arrows allow to increase or decrease the size of the object. The keyboard shortcut is `R`.
- To scale an object in a specific direction, click on one of the cubes that appears on the object and drag it to your desired size.
- If you click on the middle of the object, it scales proportionally in all directions.
- Be careful when scaling not to invert the object. When you reduce the size of the object, you can actually increase the size, but the object will be inverted. This is a huge problem when you play your game.

- You can also use the `Inspector` to adjust the Position, Rotation, and Scale of the object.
- You can select multiple objects by holding `Shift` and selecting the objects. This allows you to apply the same edit to multiple objects.
- To duplicate an object, select it and hit `Ctrl + D`.
- To delete and object, select it and hit `Ctrl + Delete`.

1.8 Adding texture and materials

- Right click on an empty space of the `Assets` folder, `Create`> `Folder`, and name it `Textures`.
- Right click on an empty space of the `Assets` folder, `Create`> `Folder`, and name it `Materials`.
- Double click on `Textures`.
- Click and drag the textures you want to use into this folder.
- Go to the `Materials` folder.
- Right click on an empty space and select `Create`> `Material`.
- Drag it to an object.
- On the `Inspector`, drag and drop a texture on the blank square.
- You can modify settings of the texture on the `Inspector` by changing the `Tiling` and `Offset` of the texture in the `X` and `Y` axis.
- `Tiling` = the number of times you want to repeat the original texture on the object.
- `Offset` = shifts the position of the texture within the object. Allows values from 0 to 1.
- `Shader` defines how the object will react to light.

1.9 Creating a tree

- Right click on an empty space of the `Assets` folder, `Import Package`> `Tree Creator`.
- The `Importing package` window pops up.
- Select all the packages and click `Import`.
- A new folder is created called `Standard Assets`.
- Go to `Tree Creator`> `Trees`.
- Drag the `BigTree` file and place it on your scene.
- On the top left corner of the screen, to the right of the object manipulator buttons, select `Pivot` to manipulate the tree.
- You can use the `Inspector` to change the position, rotation, and scale of the tree.

1.10 Lights

- There are four types of lighting: `Directional`, `Point`, `Spotlight`, and `Area`.
- To add light to the scene, go to `GameObject`> `Create Other`> and select the type of light you want.
- `Directional light` is similar to our sun, it illuminates the entire scene.
- On the `Inspector` you can adjust various settings[1]. Note that some require Unity Pro.
- You can have `Hard Shadows` with `Directional light` in Unity Free.
- Go to `Edit`> `Project Settings`> `Player` and uncheck `Use Direct3D 11`. Note that this may cause the Unity Editor to crash.
- On the `Inspector` select `Hard Shadows`. You can now adjust various settings.
- `Point light` is similar to a lamp in your living room. There is a central source of light that illuminates in all directions.
- `Spotlight` is similar to a spotlight in a theater. It illuminates a specific point.

[1] Go to `http://unity3d.com/learn/documentation` to find more information in the Unity Manual and Scripting API.

- `Area light` requires Unity Pro[2].
- `Ambient Light` is the global color of light in the scene when there are no lights present.
- To modify `Ambient Light` click `Edit`> `Render Settings`.

1.11 Particles

- Right click on an empty space of the `Assets` folder, `Import Package`> `Particles`.
- The `Importing package` window pops up.
- Select all the packages and click `Import`.
- Under `Standard Assets`> `Particles`, you can find the particles you imported.
- Drag any particle and place it in your scene.

1.12 Game controller

- Right click on an empty space of the `Assets` folder, `Import Package`> `Character Controller`.
- The `Importing package` window pops up.
- Select all the packages and click `Import`.
- Under `Standard Assets`> `Character Controllers`, you can find the files for controlling a character.
- Drag the character you want to use and place it on your scene.

1.13 Build game

- Go to `File`> `Build Settings`.
- Select the target platform.
- Click `Build`.
- Name it and select where you want to save it.

[2]See footnote 1.

Chapter 2

Ball Rolling Game

2.1 Rigidbody

- This component allows objects to have physical properties such as movement, mass, and gravity.
- On the `Inspector`, click `Add Component`> `Rigidbody`.
- Make sure all objects have `Box Collider` selected on the `Inspector`. Otherwise, objects will behave as if they were transparent.

2.2 Scripting

- This component gives some behavior to objects.
- On the `Inspector`, click `Add Component`> `New Script` and give it a name. Select `JavaScript` for the language and click `Create and Add`.
- Double click the script to open it on `MonoDevelop`.
- `MonoBehaviour` is the base class every script derives from. When you use `JavaScript`, every script automatically derives from `MonoBehaviour`. If you use `C#` or `Boo` you have to explicitly derive from `MonoBehaviour`.
- `#pragma strict` is the first directive you must put on your file. It optimizes your game and tells the compiler how to behave.
- `Start()` is called when you start the game. Usually used to set the level and finding the player.

- `Update()` is called every time the computer draws a frame. Used to check and respond inputs and user controls.
- `private var` are variables usually used for storing states that are not visible outside the script.
- `static var` are variables that can be accessed from outside the script. However, they are invisible in Unity, meaning that you cannot see it on the `Inspector`.
- `var name : type` is a variable of a given type.

2.3 BallControl.js API

- This scrips makes the ball roll left and right.
- `Input.GetAxis(axisName: string)` allows the player to control a specific axis with the arrow keys.
- `Time.deltaTime` is the time in seconds it took to complete the last frame.
- `Vector3.back` is shorthand for writing Vector3(0, 0, -1).
- `Rigidbody.AddRelativeTorque(torque: Vector3)` adds torque to this rigidbody relative to the rigidbody's own coordinate system.
- `Input.GetKeyDown(name: string)` checks if a specific key is pressed.
- `KeyCode` directly maps a physical key on the keyboard.
- `OnCollisionStay()` checks if the rigidbody is touching another rigidbody.
- `OnCollisionExit()` checks if the rigidbody has stopped touching another rigidbody.
- `Collider.Bounds.Extents.y` returns the distance from the center of the collider to the farthest point on the y axis.
- `OnCollisionEnter()` checks if the rigidbody has begun touching another rigibody.
- `Physics.Raycast(origin: Vector3, direction: Vector3, distance: float)` returns if the ray intersects a collider. The ray starts at the origin, with a given direction and distance.

2.4 CameraControl.js API

- This script makes the main camera focus and follow the Ball object.
- `transform` gets the position, rotation, and scale of this object.
- `Transform.LookAt(target: Transform)` rotates the camera so that it is always focused on the target.
- In Unity, drag the Ball object onto the `Target` variable on the `Inspector`.

2.5 Skyboxes

- These are textures for the background of the scene.
- Right click on an empty space of the `Assets` folder and select `Import Package`> `Skyboxes`.
- Import all packages.
- Select the main camera and on the `Inspector` click `Add Component`> `Rendering`> `Skybox`.
- Drag one of the skybox textures onto the `Custom Skybox`.

2.6 Prefabs

- These are duplicates of an object that are all linked to each other. A change in one of them will affect all prefabs.
- Right click on an empty space of the `Assets` folder and select `Create`> `Prefab`.
- Rename it and drag an object onto the prefab.

2.7 BallHealth.js API

- This script re spawns the Ball object after it dies.
- `Application.Load Level(name: string)` loads the passed scene.

- `Audio Source.Play()` plays the audio clip.
- `yield Microseconds(time : int)` stops the execution for the specified time.
- `Audio Source.clip.length` returns the length of the clip.
- `audio.pitch` sets the pitch of the audio clip.

2.8 Quality Settings

- This allows the user to change the quality of the images in the scene.
- To change this settings go to `Edit`> `Project Settings`> `Quality`.
- You can use the `Inspector` to change various settings[1].
- To change the default value for each platform click on the down arrow below each platform and the select the desired quality.

2.9 Collectibles

- These are objects that the Ball object will go through making, them disappear.
- To make an object go through another object without colliding, unchecked the `Collider` box on the `Inspector`.
- If you want something to happen when an object collides with another, check the `Collider` box and the `is Trigger` box. The latter is used for triggering events and the object is ignored by the physics engine.

2.10 Parent-Child Relationship

- To make an object the child of another object, drag an object from the `Hierarchy` and place it on top of the desired parent object.
- This makes the child's `Transform` variables relative to the parent's `Transform` variables.

[1] Go to `HTTP://unity.com/learn/documentation` to find more information in the Unity Manual and Scripting API.

2.11 Coin Pick.js API

- This script makes the coin disappear whenever the Ball object touches it.
- `Nonbelligerent(info: Collider)` is a trigger that executes whenever a collider hits a trigger.
- `Destroy(obj: Object, time: float)` destroys the objects after the specified time.
- `Instantiate(original: Object, position: Vector3, rotation: Quaternary)` creates an instance of the object at a position with a rotation.

2.12 Animation

- We attached an animation to the Coin object making it rotate when we play the game.
- On the `Inspector`, click `Add Component`> `Miscellaneous`> `Animation`.
- Select `Window`> `Animation`.
- Select `Add Curve`, give it a name, and select the settings you want to control.
- Drag the red line to a time frame and change one or multiple variables.
- Select the animation on the `Project` folder.
- Drag the animation to the `Animation` component on the `Inspector`.

2.13 Particles

- We attached particles to the Coin that appear only when the Coin object is touched.
- Select `Game Object`> `Create Other`> `Particle System`.
- To avoid having many particles after they have executed, make sure to destroy them in a script.

2.14 Score

- We kept score when picking up coins.
- Select `Game Object`> `Create Empty`.
- Add a script to control the score.

2.15 Game Master.js API

- This script controls the score in the game.
- `onGUI()` renders and handles GUI events. It runs twice per frame.
- `Screen.height` returns the height of the game window. It is Read Only.
- `Screen.width` returns the width of the game window. It is Read Only.
- `Sect(x: int, y: int, width: int, height: int)` creates a 2 rectangle at a position with a certain width and height.
- `GUI.Box(position: Sect, text: string)` creates a box at a position with a text on the top.
- `Game Object.FindWithTag(tag: string)` returns a gameObject with the given tag.
- `DontDestroyOnLoad(target: Object)` persists the object between scenes.

2.16 Sound

- Added sound to the game and multiple actions within in it.
- Import audio files by dragging then into Unity.
- While selecting the `CoinEffect`, select `Add Component`> `Audio`> `Audio Source`.
- Under `Audio Clip` select the clip you want to add and select `Play On Awake` to play the clip as soon as the object is instantiated.
- Remove the `Audio Listener` from the Camera object and add it to the Ball object to listen to more realistic sounds.

Chapter 3

Urban Tennis

3.1 Introduction

In this chapter, we use the tools learned from the previous chapters to develop a 2D game called "Urban Tennis". This game has a ball moving around destroying bricks by hitting them. The user controls a paddle that hits the ball to keep the game going. The player wins when they destroy all the bricks in the scene. If the player is not able to hit the ball with the paddle and it goes beyond the reach of the player, the player loses a life.

3.2 Setup

The camera `Projection` we used for all of the scenes was `Orthographic` to simulate a 2D environment. We also added a directional light to illuminate the entire scene. A plane was used as the background, cubes as bricks, walls, and paddle, and a sphere as the game ball. All objects had materials applied to them and each material had a texture applied to it. All gameObjects were prefabs to edit them more easily.

3.3 Ball Object

This gameObject had a `Rigidbody` to make it have physical properties. We also attached a `Physic Material` to add further physical properties to the object. We used it specifically to control how much the ball bounces when hitting the paddle and bricks. We set `Bounciness` to 1 to make it bounce without losing any energy when it hits another object. We also set its `Bounce combine` to `Maximum` to further make the bounciness more perfect. Finally,

since the object might be touching the background plane, we removed all the friction from the object and set the `Friction Combine` to `Minimum`. We also set the `Physic Material` to the overall game `Physic Manager` found via `Edit`> `Project Settings`> `Physics`. Here, we also set the `Bounce Threshold` to 0 to always make it bounce regardless of how slow the object is moving.

The script we attached to the object was a written in `C#`. In the previous chapters we used `JavaScript` for our scripts. You can use either one of these two languages to develop in Unity since you can get the same results. For the remainder of this chapter, all of our scripts will be developed in `C#`.

The script of this object added an upward force to the ball when the scene started. This was done for testing purposes to check the object's behavior when interacting with other objects in the scene. This was very helpful when deciding the values for the settings previously mentioned. For the final build of the game, we removed this behavior. This script also had a method that destroyed the object and re-spawned it on top of the paddle when the player lost a life.

3.4 Paddle Object

We added a script to this object to control it with the arrow keys or `A` and `D` keys. In the `Input Manager` for the `Horizontal` control found in `Edit`> `Project Settings`> `Input`, we set the gravity to 1000 to make this object stop immediately when we stop pressing a key and the sensitivity to 1000 to make this object more responsive. This script also applied some force to the object that collides with it, in our case it is always the ball object, depending on where it hits the paddle.

This script also spawned the ball object on top of the paddle when the game started or restarted after the player lost a life. We had to make the ball object a prefab to be able to instantiate it inside this script. We assigned the ball object the same position as the paddle object with a bit of offset so that the ball was on top of the paddle. The ball object also moved along with the paddle when it had not yet been fired off. We also made the `Space bar` the default key to launch the ball.

3.5 Death Field Object

For this game, we could have checked if the ball object went beyond a certain `y` position to determine if the player had died. We used that approach in the "Ball Rolling" game, but in this game we used a cube with a trigger that fired whenever the ball collided with it. This allowed us to place this "Death field" anywhere on the scene so that if the ball collided with it, the player would lose a life. In order to achieve this, we disabled the `Mesh Renderer` so that it is not visible in the game and added a trigger in the `Box Collider` field.

We added a script to this object that checked if the ball object entered this trigger. If it did, we made a call to a method of the ball object that controlled what happens when a player dies.

3.6 Main Menu

We decided to add a main menu for this game. The menu allowed the player to start the game from the last played level or start a new game. This was achieved by creating a scenes that used a `GUILayout` with buttons that had some logic behind them. The player's preferences were saved and accessed in `PlayerPrefs` in between game sessions. We used `PlayerPrefs` to keep track of what levels the user had unlocked.

3.7 Game Manager and the Singleton Pattern

Every game that we play in our daily lives has some sort of a game manager. A game manager is in charge of executing multiple actions and keeping track of states and variables of the game. What game managers actually do vary depending on the game that you develop and how you want to implement the game. However, there are certain characteristics that all game managers should have.

First of all, we want to have only one instance of the game manager that is persistent throughout the entire the game. This means that the object will not be destroyed when new scenes are loaded, but rather the same object will continue to exist in other scenes. This allows us to have a single central class that loads and saves data, keeps track of the state of the game, knows what to do when specific actions happen in the game, and is able to handle

many more options. We were able to achieve this behavior by creating a static variable that held a reference to our game manager object. This static variable was accessible to other objects and they were able to retrieve and change attributes of the game manager. On the `Awake` method, which is called when the script is loaded, we added behavior to make sure that we have only one instance of the game manager object. If the static variable was `null`, we assigned a reference of the game manager object to that variable and called the `DontDestroyOnLoad()` method on the game manager's gameObject to make it persist throughout the entire game. If the static variable was not null and it held a reference to a gameObject that was different from the gameObject that executed the script, it destroyed that gameObject since that meant that there was already a Game Master object and we did not need another one.

This pattern is similar to the Singleton Pattern used in object oriented programming and design since we only have a single instance of an object that is persistent throughout the entire game. One difference between this pattern and the Singleton Pattern is that we placed a prefab of the game manager in each scene of the game. Although our `Awake` method made sure that the game manager is not destroyed when switching between scenes, it only executed that behavior when there was a game manager in that scene. In all other scenes where there was not a game manager, the script did not execute. This is the reason why we destroyed a gameObject that did not match with the reference that the static variable was holding. When a new scene was loaded and a game manager was coming from that scene, there were two game manager objects in that scene: the one from the previous scene and the prefab placed before the game started executing. When the prefab game manager's script executed, it knew that there was already a game manager from another scene and it destroyed itself. This was a very useful approach since it allowed us to edit and debug any scene we wanted since there was always a game manager present in every single. Otherwise, if a game manager was only placed on one scene, we would not have a game manager in other scene and we would have to always start the game from scene containing the game manager and walk our way through the entire game until we reached a scene that we wanted to work with.

3.8 Data Persistency

By using the Singleton Pattern, we were able to persist the game data between scenes. There was one scene where the game manager was created and all the information and settings that occurred during that scene were persisted and carried to the next scene. However, we also wanted the game data to persist between executions of the game. Although our game manager was not destroyed when loading a new scene, it was destroyed, along with all the information of the game, when we exited our game. In order to preserve information and the game state between executions of the game, we decided to save the information to a local file before quitting the game and load it when the game started.

We achieved this by creating a serialized `C#` class without extending the `MonoBehaviour` from Unity. We used instance variables to store the information we wanted to persist between executions. We also used getters and setters to retrieve and modify those fields. The class had to be serialized in order to be able to write it to a binary file.

In order to save the information we used `C#` commands and methods to serialize an object of the information we wanted to persist before quitting the game. Serialization is the process of transforming an object to a format that can be stored in file. In order to load the information, we also used `C#` commands and methods for deserializing the binary file stored in our local computer as the `C#` object that contained the information we wanted to persist.

Although Unity's `PlayerPrefs` persists information you save there between game sessions, the method discussed above allows you to have more flexibility to what you want to save. `PlayerPrefs` only allows you to save strings, ints, and floats, whereas saving to a file allows you to persist more complicated data structures and other information of the game.

Chapter 4

Technologies

4.1 Unity

Unity is a game creating system that comes with a game engine, the Unity editor, and an integrated development environment (IDE) that allows developers to create games for different platforms. You can either buy Unity Pro or download the free version with some reduced features at `http://unity3d.com/unity/download`.

4.2 MonoDevelop

Monodevelop is an IDE for scripting in JavaScript, C#, and Boo. It offers many features such as integration with Unity, code completion, syntax highlighting, debugging tools for Unity, and many more. When you either buy or download the free version of Unity, MonoDevelop is also downloaded and synced with the Unity editor.

4.3 VisualSVN Server

VisualSVN Server is an Apache Subversion Server that provides a Subverion version control system for Windows computers. It allows the user to create, host, and manage local SVN repositories for version control. We use VisualSVN Server to create, host, and manage repositories for our games. You can download the free version at `http://www.visualsvn.com/server/`.

4.4 TortoiseSVN

TortoiseSVN is a Apache Subversion client that provides a user interface for Subversion commands and tasks in Windows computers. We use TortoiseSVN to perform initial commits of our games since version control in Unity is a feature of Unity Pro only. You can download this software at `http://tortoisesvn.net/`.

4.5 Eclipse and Subclipse

Eclipse is an IDE for multiple programming languages. Eclipse allows the user to install custom plug-ins to add additional features to the IDE. One of these plug-ins is Subclipse which provides support for Subversion. One of the features of Subclipse is SVN Repository Exploring, which allows the user to manage SVN repositories. It also allows the user to perform Subversion commands within Eclipse. We use Eclipse along with Subclipse to do all of our version control commands such as committing changes, updating our code, reverting to previous versions, and much more. You can get Eclipse at `http://www.eclipse.org/downloads/` and download the Subclipse plug-in from the Eclipse Marketplace.

4.6 YouTrack

YouTrack is an issue tracking and project managing tool. You can buy licenses for multiple users to store all of your information on their server. You can also download the stand alone application to host the information on your own computer or server and have an unlimited number of users accessing it. We use the stand alone version of YouTrack to manage our projects and track issues and features of our games. You can find YouTrack at `http://www.jetbrains.com/youtrack/`.

4.7 Version Control

As mentioned before, version control within Unity is only available for Unity Pro accounts. There are many workarounds for Unity Free developer to have version control for their

games. The following is the procedure we use to have version control for our games using the technologies previously described.

- Using VisualSVN Server, create a repository.
- Create a new project in your repository.
- Go into Unity, create a basic empty Unity project, go to `Edit`> `Project Settings`> `Editor` to enable meta files, and save the project.
- Import the Unity project to the trunk folder of the project in the repository using TortoiseSVN.
- Using VisualSVN Server, go into the imported project location and delete the Library folder.
- Use Eclipse to check out the project from the repository to your workspace using the New Project Wizard.
- Select General Project under the wizard's menu.
- Open the new project in your workspace using Unity.
- Use Eclipse to commit and update your game.

The only file and folder that you should commit are the .project file generated by Eclipse and the Assets and ProjectSettings folders. The rest of folders and files generated by Unity can be regenerated by selecting `Assets`> `Sync MonoDevelop Project`.