6.1 Project Optimization

Techniques:

Length:	30 minutes
Overview:	In this lesson, you will learn about a variety of different techniques to optimize your projects and make them more performant. You may not notice a huge difference in these small prototype projects, but when you're exporting a larger project, especially one for mobile or web, every bit of performance improvement is critical.
Project Outcome:	Several of your prototype projects will have improved optimization, serving as examples for you to implement in your personal projects
Learning Objectives:	By the end of this lesson, you will be able to: - Recognize and use new variable attributes to keep values private, but still editable in the inspector - Use the appropriate Unity Event Functions (e.g. Update vs. FixedUpdate vs. LateUpdate) to make your project run as smoothly as possible - Understand the concept of Object Pooling, and appreciate when it can be used to optimize your project

1: Variable attributes

In the course, we only ever used "public" or "private" variables, but there are a lot of other variable attributes you should be familiar with.

1. Open your **Prototype 1** project and open the **PlayerController.cs** script
2. Replace the keyword "private" with **[SerializeField]**, then edit the values in the inspector
3. In **FollowPlayer.cs**, add the **[SerializeField]** attribute to the Vector3 **offset** variable
4. Try applying the "**readonly**", "**const**", or "**static**" attributes, noticing that all have the effect of removing the variable from the inspector

- **New Concept:** using [SerializeField] instead of public attribute
- **Tip:** "protected" is very similar to "private", but would also allow access to derived classes

```
[SerializeField] ~~private~~ float speed = 30.0f;
[SerializeField] ~~private~~ float turnSpeed = 50.0f;

[SerializeField] ~~private~~ Vector3 offset = new Vector3(0, 5, -7);
```

2: Unity Event Functions

In the course we only ever used the default Update() and Start() event functions, but there are others you might want to use in different circumstances

1. **Duplicate** your main Camera, rename it "Secondary Camera", then **deactivate** the Main Camera
2. **Reposition** the Secondary camera in a first-person view, then edit the **offset variable** to match that position
3. Run your project and notice how choppy it is
4. In **PlayerController**.cs, change "Update" to "FixedUpdate", and in **FollowPlayer**.cs, change "Update" to "LateUpdate", then **test again**
5. **Delete** the Start() function in both scripts, then reactivate your Main Camera

- **New Concept:** "Event Functions" are Unity's default methods that run in a very particular order over the life of a script (e.g. Start and Update)
- **New Concept:** Update vs FixedUpdate vs LateUpdate
- **New Concept:** Awake vs Start
- **Tip:** If you're not using Start or Update, it's cleaner to delete them

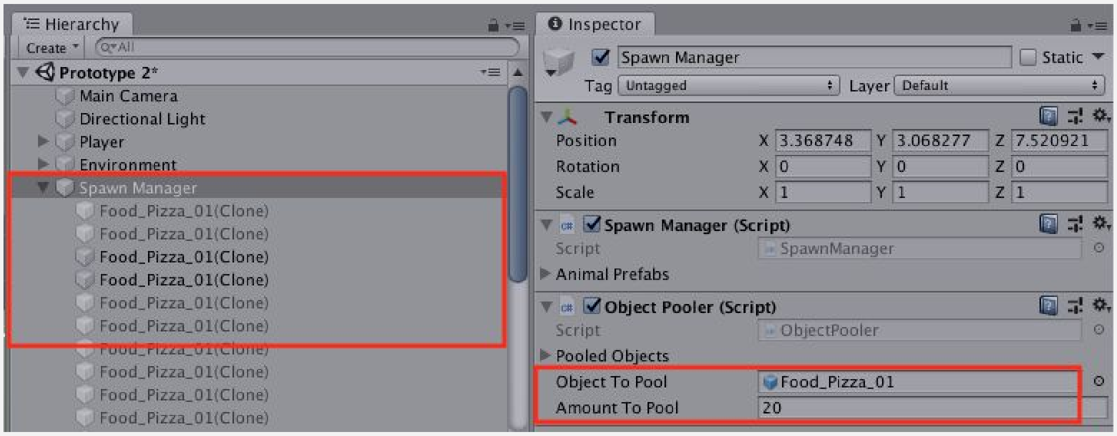
```
PlayerController.cs
void FixedUpdate() { ...

FollowPlayer.cs
void LateUpdate() { ...
```

3: Object Pooling

Throughout the course, we've created a lot of prototypes that instantiated and destroyed objects during gameplay, but there's actually a more performant / efficient way to do that called Object Pooling.

1. Open **Prototype 2** and create a backup
2. **Download** the **Object Pooling** unity package and **import** it into your scene
3. Reattach the **PlayerController** script to your player and reattach the **DetectCollisions** script to your animal prefabs (***not*** to your food prefab)
4. Attach the **ObjectPooler** script to your Spawn Manager, drag your projectile into the "**Objects To Pool**" variable, and set the "**Amount To Pool**" to 20
5. **Run** your project and see how the projectiles are activated and deactivated

- **Warning**: You will be overwriting your old work with this new system, so it's important to make a backup first in case you want to revert back
- **New Concept**: Object Pooling: creating a reusable "pool" of objects that can be activated and deactivated rather than instantiated and destroyed, which is much more performant
- **Tip**: Try reading through the new code in the ObjectPooler and PlayerController scripts
- **Don't worry**: If your project is small enough that you're not experiencing any performance issues, you probably don't have to implement this



Prototype 2*
Main Camera
Directional Light
Player
Environment
Spawn Manager
Food_Pizza_01(Clone)
Food_Pizza_01(Clone)
Food_Pizza_01(Clone)
Food_Pizza_01(Clone)
Food_Pizza_01(Clone)
Food_Pizza_01(Clone)
Food_Pizza_01(Clone)
Food_Pizza_01(Clone)
Food_Pizza_01(Clone)
Food_Pizza_01(Clone)

☑ Spawn Manager
Tag Untagged Layer Default
Position X 3.368748 Y 3.068277 Z 7.520921
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
☑ Spawn Manager (Script)
Script SpawnManager
☑ Object Pooler (Script)
Script ObjectPooler
Object To Pool Food_Pizza_01
Amount To Pool 20

Lesson Recap

New Concepts and Skills

- Optimization
- Serialized Fields
- readonly / const / static / protected
- Event Functions
- FixedUpdate() vs. Update() vs. LateUpdate()
- Awake() vs. Start()
- Object Pooling

6.2 Research and Troubleshooting

Steps:

Step 1: Make the vehicle use forces

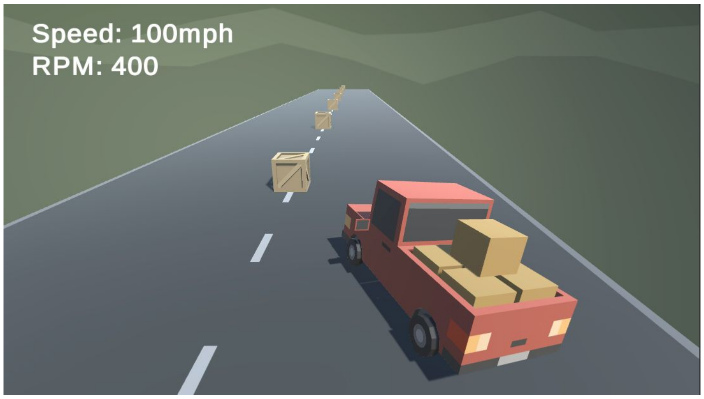
Step 2: Prevent car from flipping over

Step 3: Add a speedometer display

Step 4: Add an RPM display

Step 5: Prevent driving in mid-air

Example of project by end of lesson



Length: 75 minutes

Overview: In this lesson, you will attempt to add a speedometer and RPM display for your vehicle in Prototype 1. In doing so, you will learn the process of doing online research when trying to implement new features and troubleshoot bugs in your projects. As you will find out, adding a new feature is very rarely as simple as it initially seems - you inevitably run into unexpected complications and errors that usually require a little online research. In this lesson, you will learn how to do that so that you can do it with your own projects.

Project Outcome: By the end of this lesson, the vehicle will behave with more realistic physics, and there will be a speedometer and Revolution per Minute (RPM) display.,

Learning Objectives: By the end of this lesson, you will be able to:

- Use Unity Forums, Unity Answers, and the online Unity Scripting Documentation to implement new features and troubleshoot issues with your projects

Step 1: Make the vehicle use forces

If we're going to implement a speedometer, the first thing we have to do is make the vehicle accelerate and decelerate more like a real car, which uses forces - as opposed to the Translate method.

1. Open your **Prototype 1** project and make a backup
2. Replace the Translate call with an AddForce call on the vehicle's Rigidbody, renaming the "speed" variable to "horsePower"
3. Increase the ***horsePower*** to be able to actually move the vehicle
4. To make the vehicle move in the appropriate direction, change AddForce to AddRelativeForce

- **New Concept:** using Unity Documentation
- **New Concept:** using Unity Answers
- **New Concept:** AddRelativeForce
- **Don't worry:** Still a big issue where the vehicle can drive in air and that it flips over super easily!

```
private Rigidbody playerRb;

void Start() {
  playerRb = GetComponent<Rigidbody>();
}

void FixedUpdate() {
  ~~transform.Translate(Vector3.forward * speed * verticalInput);~~
  playerRb.AddRelativeForce(Vector3.forward * verticalInput * horsePower);
}
```

Step 2: Prevent car from flipping over

Now that we've implemented real physics on the vehicles, it is very easy to overturn. We need to figure out a way to make our vehicle safer to drive.

1. Add wheel colliders to the wheels of your vehicle and edit their radius and center position, then disable any other colliders on the wheels
2. Create a new ***GameObject centerOfMass*** variable, then in Start(), assign the playerRb variable to the centerOfMass position
3. Create a new **Empty Child** object for the vehicle called "Center Of Mass", reposition it, and assign it to the **Center Of Mass** variable in the inspector
4. **Test** different center of mass positions, speed, and turn speed values to get the car to steer as you like

- **New Concept:** Wheel colliders
- **New Concept:** Center of Mass
- **Don't Worry**: We can still drive the vehicle when it's sideways or upside down
- **Warning:** This is still not the *proper* way to do vehicles - should actually be rotating / turning the wheels

```
[SerializeField] GameObject centerOfMass;

void Start() {
  playerRb.centerOfMass = centerOfMass.transform.position;
}
```

Step 3: Add a speedometer display

Now that we have our vehicle in a semi-drivable state, let's display the speed on the User Interface.

1. Add a new ***TextMeshPro - Text*** object for your "Speedometer Text"
2. Import the **TMPro library**, then create and assign new create a new **TextMeshProUGUI** variable for your ***speedometerText***
3. Create a new float variables for your ***speed***
4. In Update(), **calculate** the speed in mph or kph then **display** those values on the UI

- **Warning**: Will be going fast through adding the text, since we did this in prototype 5
- **New Concept:** RoundToInt

```
using TMPro;

[SerializeField] TextMeshProUGUI speedometerText;
[SerializeField] float speed;

private void Update() {
  speed = Mathf.Round(playerRb.velocity.magnitude * 2.237f); // 3.6 for kph
  speedometerText.SetText("Speed: " + speed + "mph");
}
```

Step 4: Add an RPM display

One other cool feature that a lot of car simulators have is a display of the RPM (Revolutions per Minute) - the tricky part is figuring out how to calculate it.

1. Create a new "RPM Text" object, then **create** and **assign** a new **rpmText variable** for it
2. In Update(), calculate the the RPMs using the Modulus/Remainder operator (%), then display that value on the UI

- **New Concept:** Modulus / Remainder (%) operator

```
[SerializeField] TextMeshProUGUI rpmText;
[SerializeField] float rpm;

private void Update() {
  rpm = Mathf.Round((speed % 30)*40);
  rpmText.SetText("RPM: " + rpm);
}
```

Step 5: Prevent driving in mid-air

Now that we have a mostly functional vehicle, there's one other big bug we should try to fix: the car can still accelerate/decelerate, turn, and increase in speed/rpm in mid-air!

1. Declare a new **List** of **WheelColliders** named ***allWheels*** (or frontWheels/backWheels), then assign each of your wheels to that list in the inspector
2. Declare a new ***int wheelsOnGround***
3. Write a ***bool IsOnGround()*** method that returns true if all wheels are on the ground and false if not
4. Wrap the **acceleration**, **turning**, and **speed/rpm** functionality in if-statements that check if the car is on the ground

- **New Concept:** looping through lists
- **New Concept**: custom methods with bool returns
- **Tip**: if you use frontWheels or backWheels, make sure you only drag in two wheels and only test that wheelsOnGround == 2

```
[SerializeField] List<WheelCollider> allWheels;
[SerializeField] int wheelsOnGround;

if (IsOnGround()) {[ACCELERATION], [ROTATION], [SPEED/RPM]}

bool IsOnGround () {
  wheelsOnGround = 0;
  foreach (WheelCollider wheel in allWheels) {
    if (wheel.isGrounded) {
      wheelsOnGround++;
    }
  }
  if (wheelsOnGround == 2) {
    return true;
  } else {
    return false;
  }
}
```

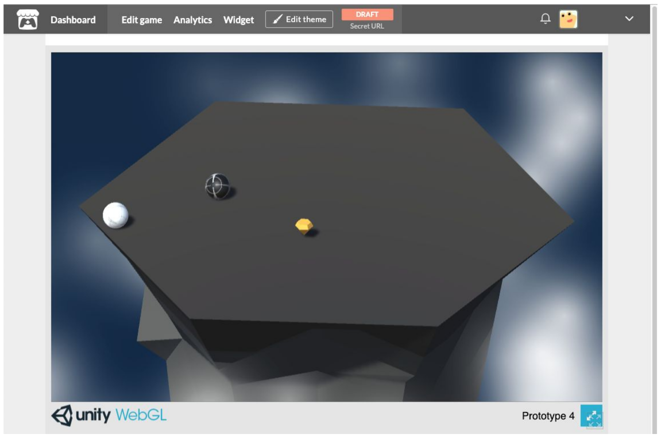
Lesson Recap

New Concepts and Skills

- Searching on Unity Answers, Forum, Scripting API
- Troubleshooting to resolve bugs
- AddRelativeForce, Center of Mass, RoundToInt
- Modulus/Remainder (%) operator
- Looping through lists
- Custom methods with bool return

6.3 Sharing your Projects

Example of project by end of lesson



Steps:

Step 1: Install export Modules

Step 2: Build your game for Mac or Windows

Step 3: Build your game for WebGL

Length: 30 minutes

Overview: In this lesson, you will learn how to build your projects so that they're playable outside of the Unity interface. First, you will install the necessary export modules to be able to publish your projects. After that, you will build your project as a standalone app to be played on Mac or PC computers. Finally, you will export your project for WebGL and even upload it to a game sharing site so that you can send it to your friends and family.

Project Outcome: Your project will be exported and playable as a standalone app on Mac/PC or for embedding online.

Learning Objectives: By the end of this lesson, you will be able to:

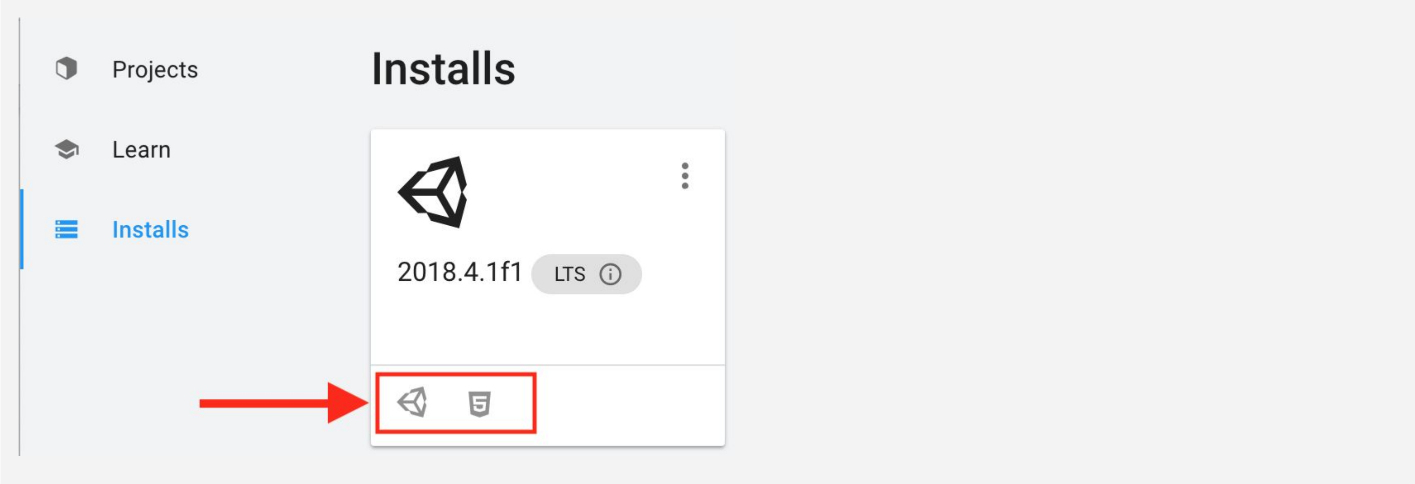
- Add and manage export modules for your Unity installs so you can choose which platforms to build for
- Build your projects for Mac or PC so they can be played as standalone apps
- Build your projects for WebGL so they can be uploaded and embedded online and shared with a single URL

Step 1: Install export Modules

Before we can export our projects, we need to add the "Export Modules" that will allow us to export for particular platforms.

1. Open **Unity Hub** and click to **Add Modules** to the version of Unity you have used in the course
2. Select **WebGL Build Support**, and either **Mac** or **Windows** build support, then click **Done** and wait for the installation to complete

- **Tip** - Mac and Windows will create apps for your computer and WebGL will allow you to publish online
- **Tip** - you should see little icons appear when it is complete
- **Tip** - WebGL is nice because you can more easily share it online and it is platform-independent

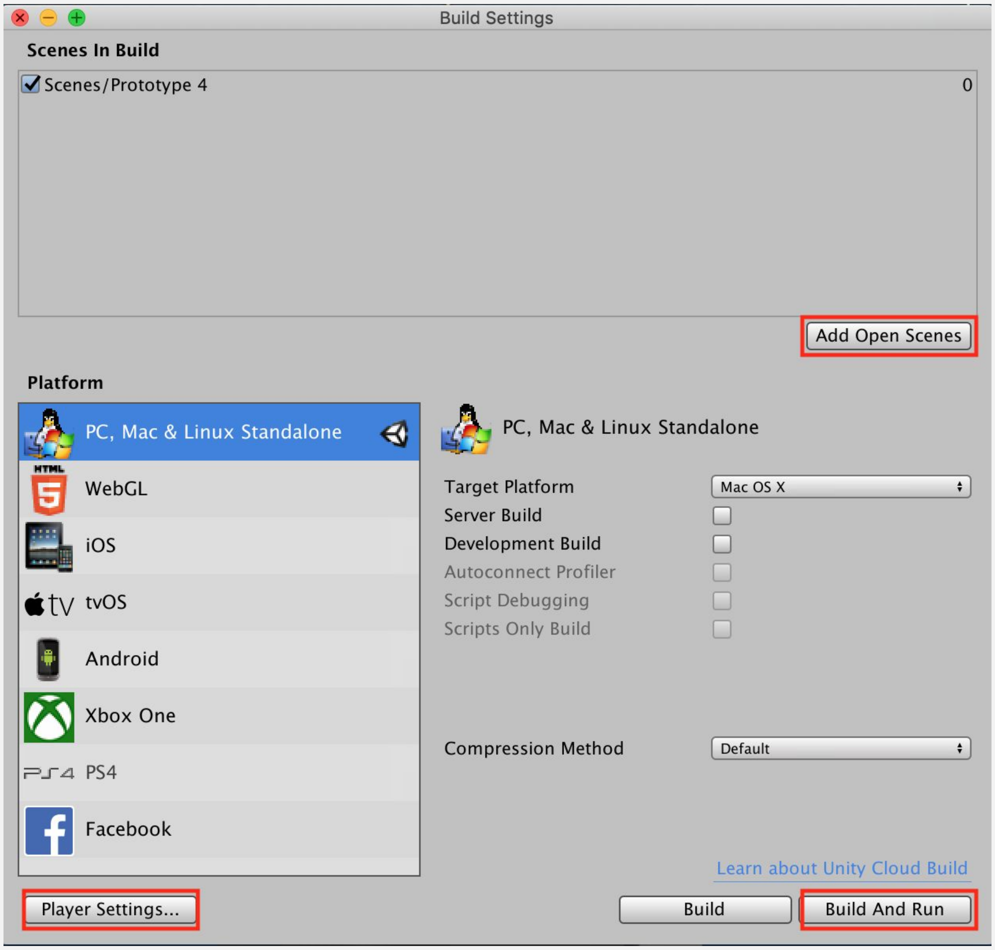


Step 2: Build your game for Mac or Windows

Now that we have the export modules installed, we can put them to use and export one of our projects

1. **Open** the project you would like to build (could be a prototype or your personal project)
2. In Unity, click **File > Build Settings,** then click **Add Open Scenes** to add your scene
3. Click **Player Settings** and adjust any settings you want, including making it "Windowed", "Resizable", and whether or not you want to enable the "Display Resolution Dialog"
4. Click **Build**, name your project, and save it inside a new folder inside your Create with Code folder called "Builds"
5. **Play** your game to test it out, then if you want, **rebuild** it with different settings

- **Don't worry** - a prototype that's not fully playable will be problematic when you share it because the user will have to close and reopen it to play it again, but that's OK for now
- **Tip** - since it's just a mini-game, it might be better to use "Windowed" - this also allows the player to more easily exit since we don't have a full UI to do that
- **Don't worry** - on Windows, you have an .exe file and a Data folder - on Mac, you just have a .app file
- **Warning** - it's kind of hard to distribute these as is because most email clients are cautious of executables like this



Build Settings

Scenes In Build

☑ Scenes/Prototype 4 0

Platform: PC, Mac & Linux Standalone

Target Platform: Mac OS X

Server Build ☐

Development Build ☐

Autoconnect Profiler ☐

Script Debugging ☐

Scripts Only Build ☐

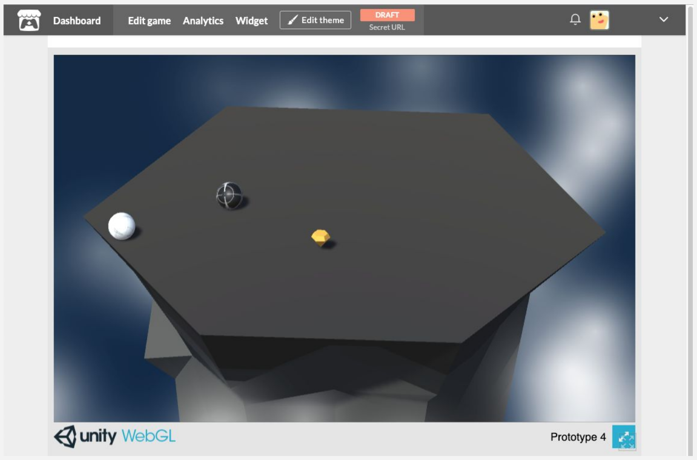
Compression Method: Default

Step 3: Build your game for WebGL

Since it is pretty hard to distribute your games on Mac or Windows, it's a good idea to make your projects available online by building for WebGL.

1. Reopen the **Build Settings** menu, select **WebGL**, then click **Switch Platform**
2. Click **Build**, then save in your "Builds" folder with "<u>- WebGL</u>" in the name
3. Try clicking on **index.html** to run your project (you may have to try opening with different browsers)
4. Right-click on your WebGL build folder and **Compress/Zip** it into a .zip file
5. If you want, **upload** it to a game sharing site like itch.io

- **Warning** - it's easy to forget to click "Switch platform" and can be confusing
- **Don't worry** - building for WebGL can take a long time
- **Warning** - some browsers do not support opening WebGL programs from your computer
- **Tip** - If uploading your game to a site like itch.io, make sure to choose "HTML" format and to "Play in browser"



Lesson Recap

New Concepts and Skills

- Installing export modules
- Building for Mac/PC
- Building for WebGL/HTML