(Week 12)

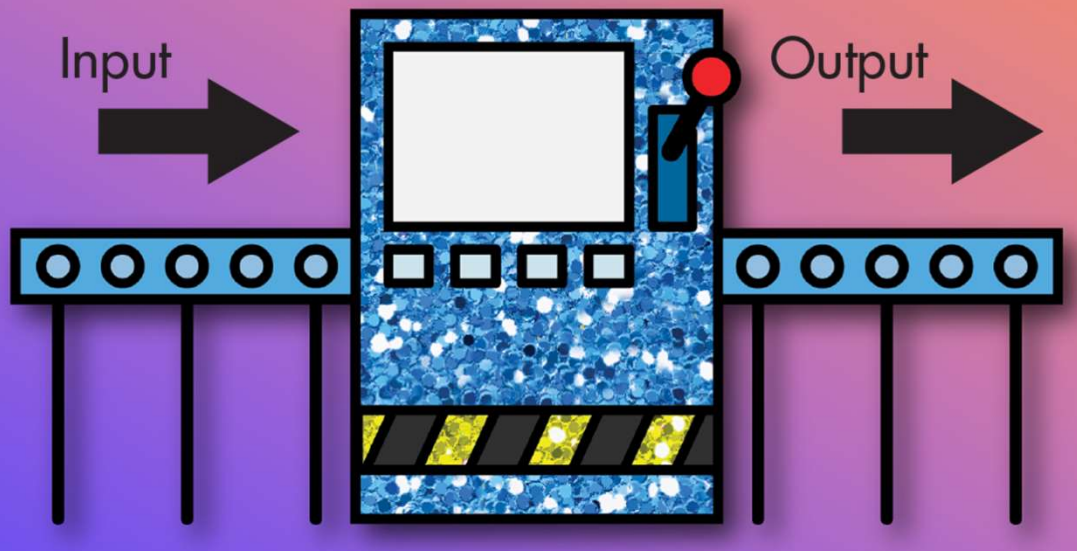
การเขียนโปรแกรมคอมพิวเตอร์ขั้นสูงเพื่อควบคุมอุปกรณ์

ADVANCE COMPUTER PROGRAMMING

สอนโดย พงศธร เกียรติเจริญพร (มิว)

Optimization

Optimization



คอมพิวเตอร์

CPU : Central Processing Unit
เปรียบได้กับสมองส่วนประมวลผล
สามารถประมวลสิ่งที่ซับซ้อนมาก ๆได้ (เช่น AI) แต่ทำงานได้ทีละอย่าง

GPU : Graphic Processing Unit
คำนวณค่าสีที่จะแสดงออกมาบนจอ (ทุก Pixel)
ไม่สามารถคิดคำนวณอะไรที่ซับซ้อนได้ แต่ทำงานหลาย ๆอย่างพร้อมกันได้

Ram : Random Access Memory
เป็นสถานที่เก็บข้อมูลชั่วคราว ก่อนที่ CPU จะนำข้อมูลเหล่านั้นไปประมวลผล

Harddrive
เก็บข้อมูลทั้งหมด เพื่อที่จะดึงออกมาประมวลผลในภายหลัง
เปรียบเสมือนสมองส่วนความจำ

05/10/2021

RAM

โหลดข้อมูลที่จำเป็น

CPU / GPU

ประมวลผลคำนวณสีที่จะแสดง

HDD

ข้อมูลทุกอย่างในเกม

Monitor

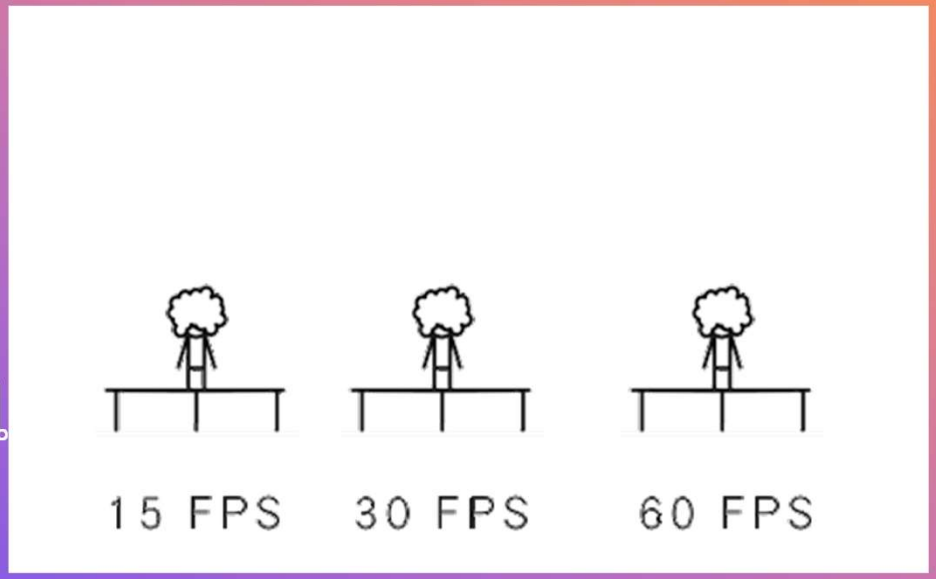
แสดงภาพ

เกม

** ทดสอบกับเครื่อง Spec ขั้นต่ำที่เกมจะ Support **
** ทดสอบที่ Resolution ต่ำสุดที่เกมจะ Support **

1. ปัญหา Framerate (CPU GPU)
- Framerate ต่ำกว่าที่ต้องการ
- Framerate วิ่งไม่คงที่

2. ปัญหา Memory (RAM HDD)
- เกมมีขนาดใหญ่เกินความจำเป็น
- กิน Ram มากผิดปกติ (อาจทำให้เกม Crash ไ
- โหลดเข้าเกมนาน



การตรวจสอบและวัดผล

การตรวจสอบ

1. ระบุปัญหา : เกมมี Framerate ต่ำ , วิ่งอยู่ที่ 40 - 45 FPS

2. ตั้งสมมติฐาน : น่าจะเกิดจาก Polygon เยอะเกินไป

3. ทดลอง : ปรับโมเดลทั้งหมดให้เป็น Low Poly --> Framerate กลายเป็น 40 -

42 FPS ดีขึ้น แต่ยังไม่น่าพอใจ

4. สรุปผล :เรื่อง Polygon มีผลเพียงบางส่วน แต่ยังไม่ใช่สาเหตุหลัก

การตรวจสอบ (ต่อ)

2. ตั้งสมมติฐาน : อาจจะเกิดจากโค้ดที่เขียน Update() ทำงานสิ้นเปลือง

3. ทดลอง : ลองเปิด Script ทีละตัว --> พบว่าตอนปิด Script A , Framerate เด้งขึ้นมาเป็น 60

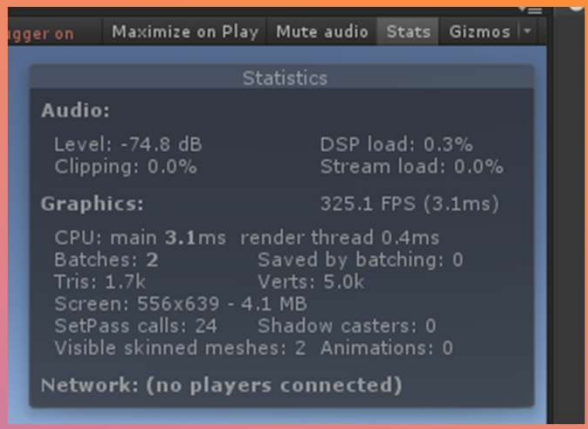
4. สรุปผล : Script A มีปัญหา , จะนำไปวิเคราะห์ปรับแก้ต่อไป

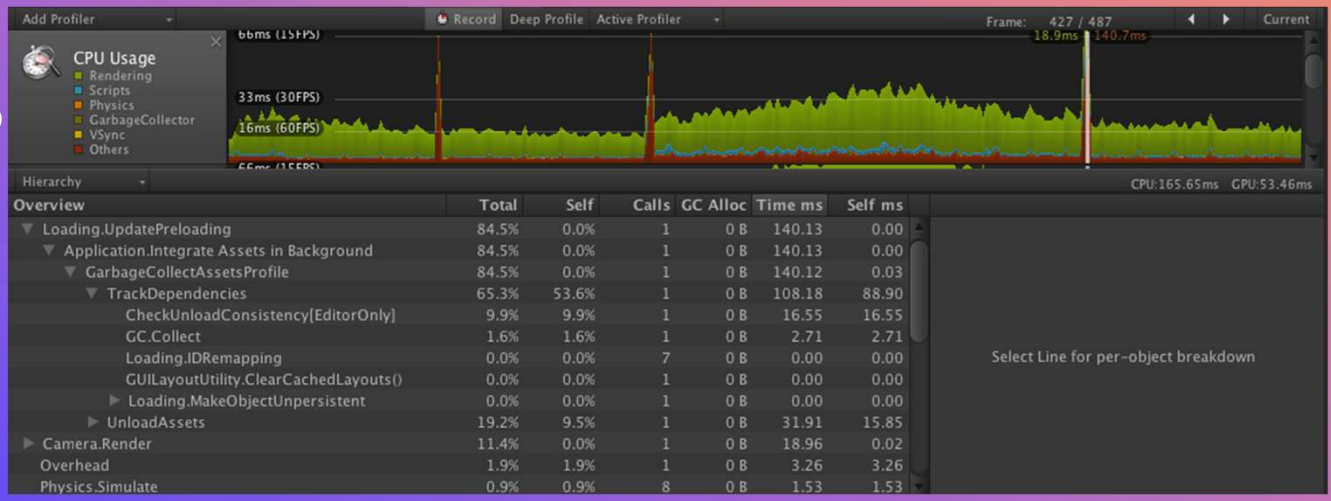
** ในขั้นตอนทดลอง ต้องควบคุมสิ่งรอบๆตัวให้เหมือนเดิมทุกการทดสอบ เช่น เครื่องที่ทดสอบเป็นเครื่องเดียวกัน , ไม่เปิด Youtube หรือโปรแกรมอื่นๆระหว่างการทดสอบแต่ละครั้ง , ทดสอบที่ Resolution เท่าเดิม , ไม่เปิด Scene View ระหว่างทำการทดสอบ **

** เกมแต่ละเกม จะมีปัญหา Performance ไม่เหมือนกัน , เราควรจะหาสาเหตุก่อนที่จะลงมือแก้ไข

การวัดผล

- ดูจาก Render Statistic Windows
- ติด Script : FPS Counter
- ดูจาก Profiler
- ดูขนาดไฟล์หลังจาก B





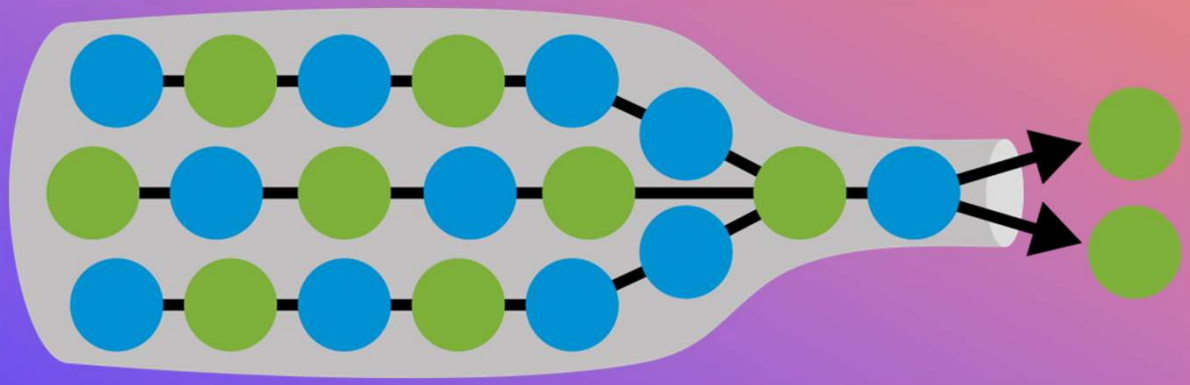
ปัญหา Framerate

- Framerate ขึ้นอยู่กับความเร็วในการวาดภาพของ GPU

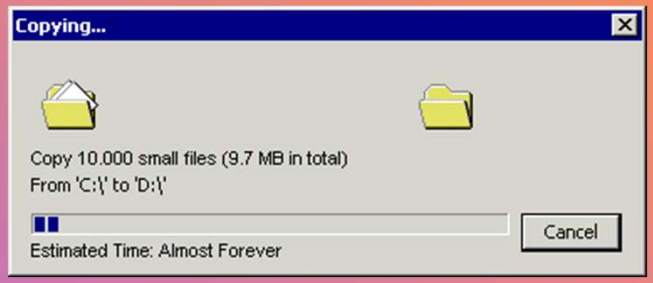
Bottleneck

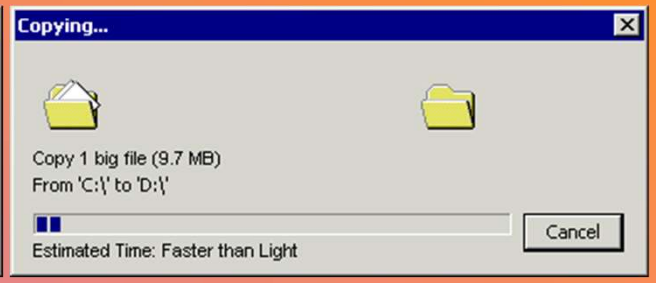
CPU เป็นคนสั่งให้ GPU วาดภาพ

- ถ้า CPU ช้า แต่ GPU เร็ว --> CPU สั่งงานช้า GPU มีเวลาว่างเยอะ ได้ภาพน้อย --> Framerate ต่ำ
- ถ้า CPU เร็ว แต่ GPU ช้า --> CPU สั่งงานเร็ว GPU ทำงานช้าวาดภาพไม่ทัน --> Framerate ต่ำ



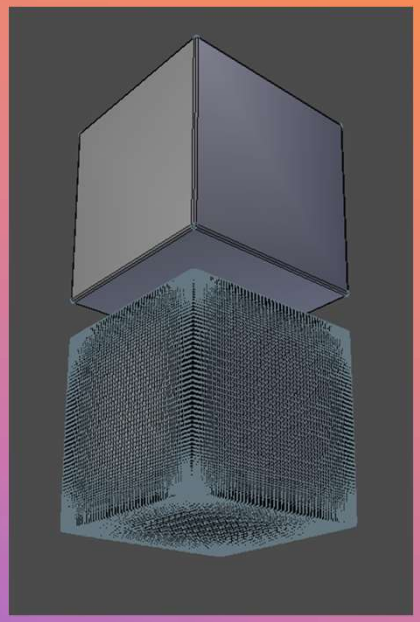
ปัญหา Framerate





CPU

- Process แต่ละ Frame หนักเกินไป
  - --> โค้ด Update() ซับซ้อน
  - --> Physics System (Collision , 2D , 3D , Layer , etc.)
  - --> Particle System
- Drawcall เยอะเกินไป --> Texture Atlas



GPU

- กราฟฟิคการ์ด คำนวณการแสดงผลไม่ทัน
  - -> Shader มีความซับซ้อนมากเกินไป --> ปรับ Shader
  - -> Polygon มากเกินไป --> LOD , Normal Map , Optimize Model
  - -> คำนวณแสงมากเกินความจำเป็น --> Lightmap
  - -> จำนวน Pixel มากเกินไป --> ลด Quality ลด Resolution
  - -> มีสิ่งที่ต้องเขียนบนจอ มากเกินไป (Overdraw) --> ลดความซับซ้อนของฉาก , Occlusion Culling

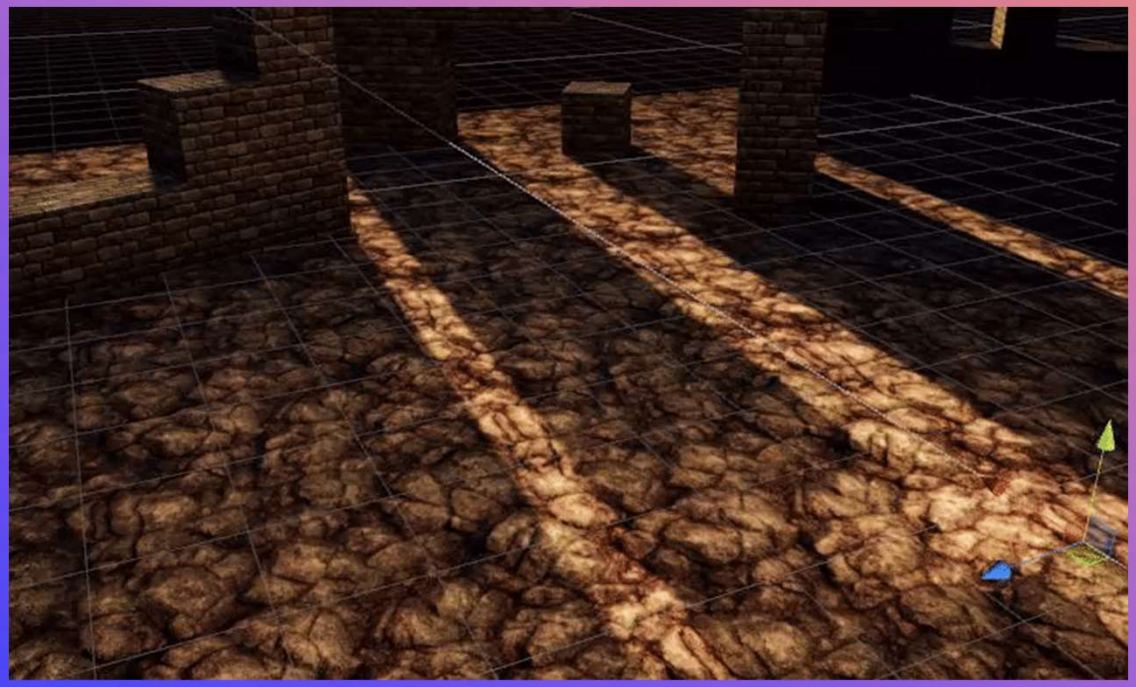
Framerate ไม่คงที่

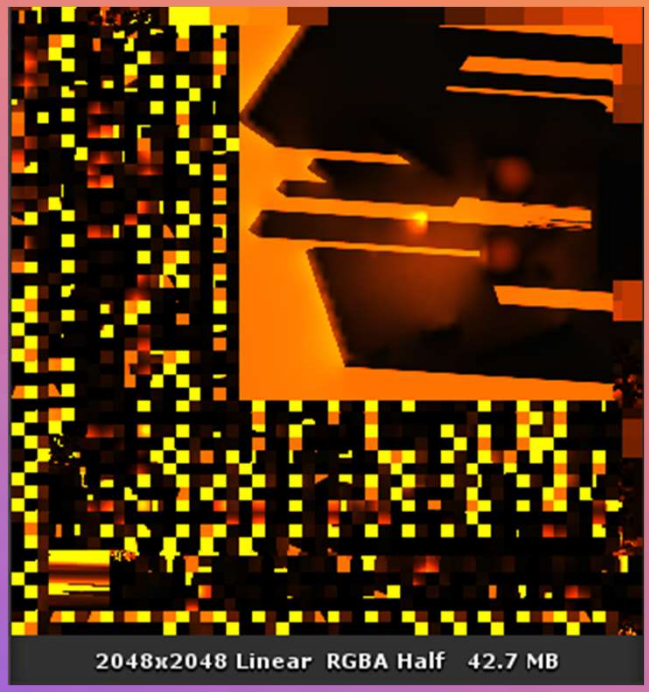
- ในระหว่างการเล่นเกม จะมีจุดที่เกมทำงานหนักที่สุด (Fully Load) ซึ่งอาจทำให้ Framerate ลดลงอย่างรุนแรง เช่น เกิดเอฟเฟคระเบิดตึกถล่ม , ตอนทีมไฟท์
- เป็นหนึ่งในสาเหตุที่สร้างความรำคาญให้กับประสบการณ์การเล่นเกมอย่างมาก
- เราควรทดสอบและ Optimize เกมของเราในขณะ Fully Load ไม่ให้ Framerate ลดลงมากเกินไปจนรู้สึกได้ หรือทำการ Limit Framerate วิ่งคงที่ที่ระดับขั้นต่ำ เช่น 30 FPS

Lightmap

GPU ↓↓↓

Ram / Disk ↑↑↑

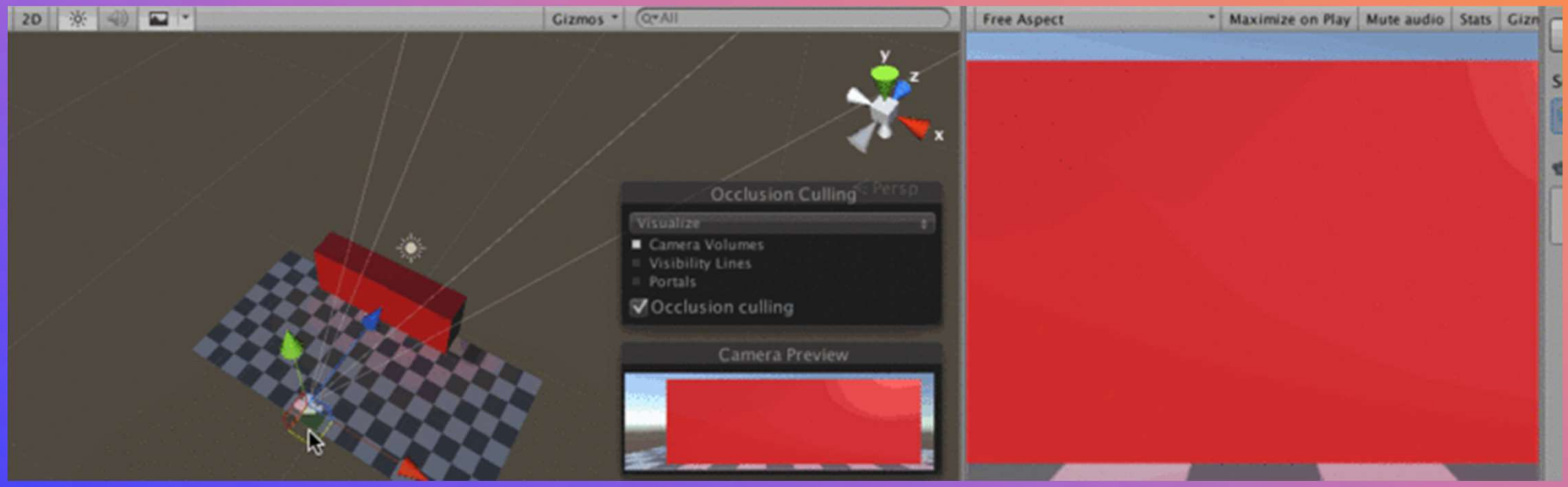




Occlusion Culling

GPU ↓↓↓

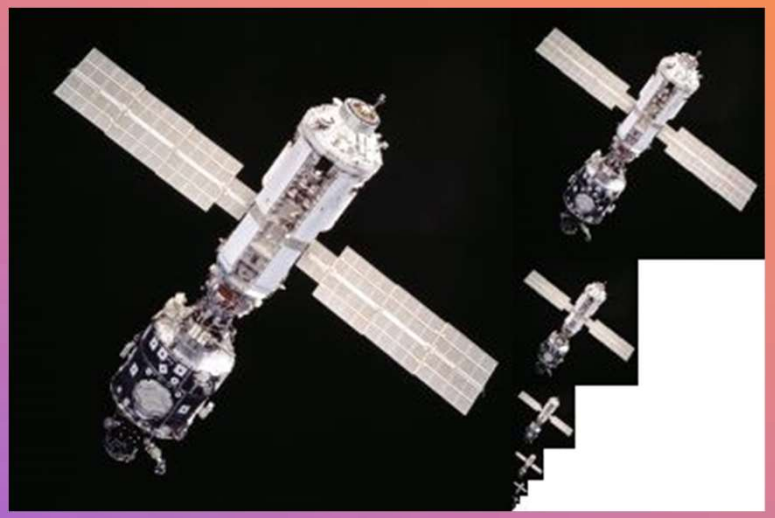
CPU ↑



Mipmap

GPU ↓

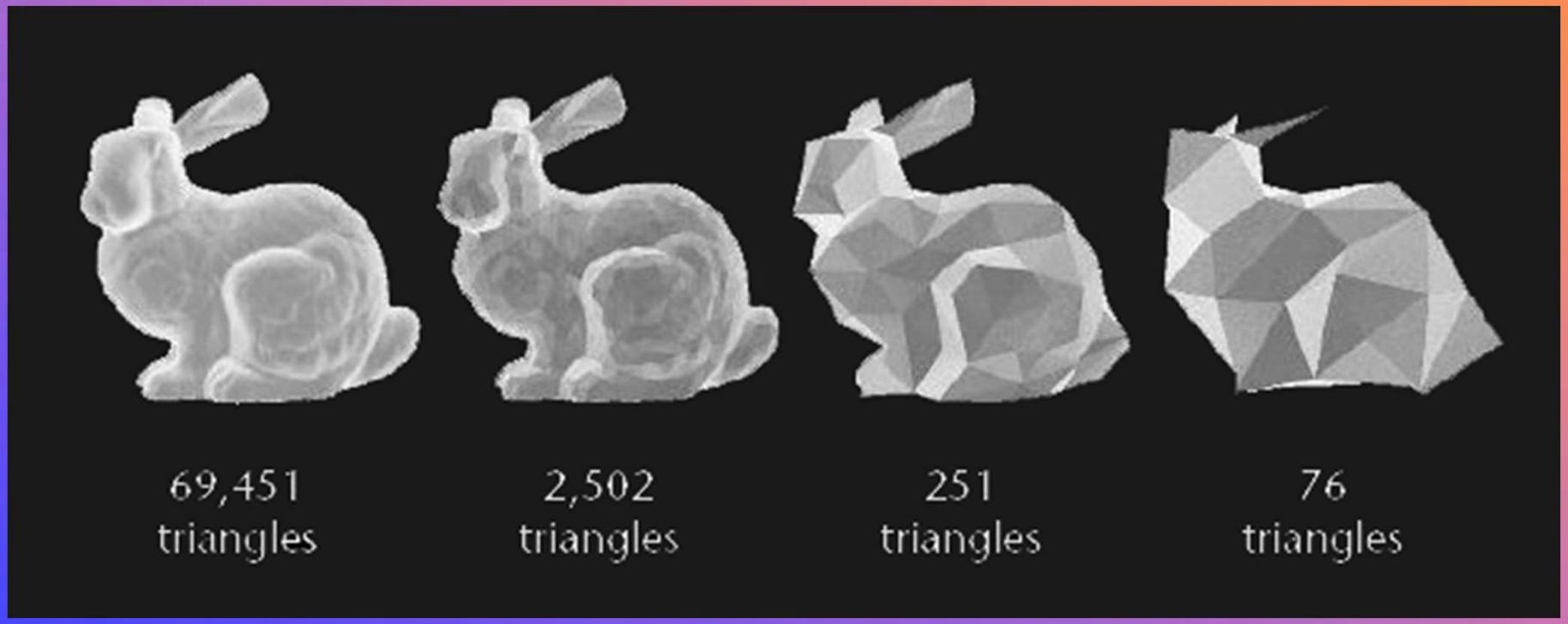
Ram / Disk ↑



Level of Detail

GPU ↓↓↓

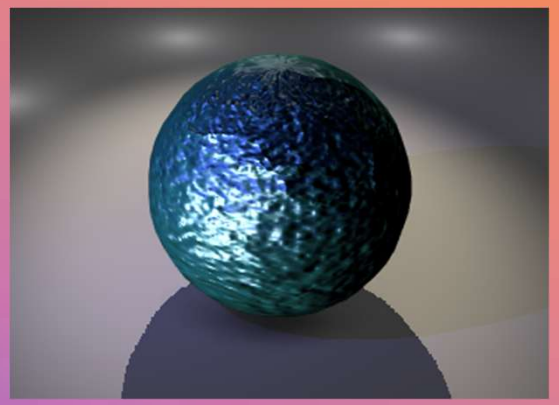
Ram / Disk ↑



Normal Map

GPU ↓↓↓

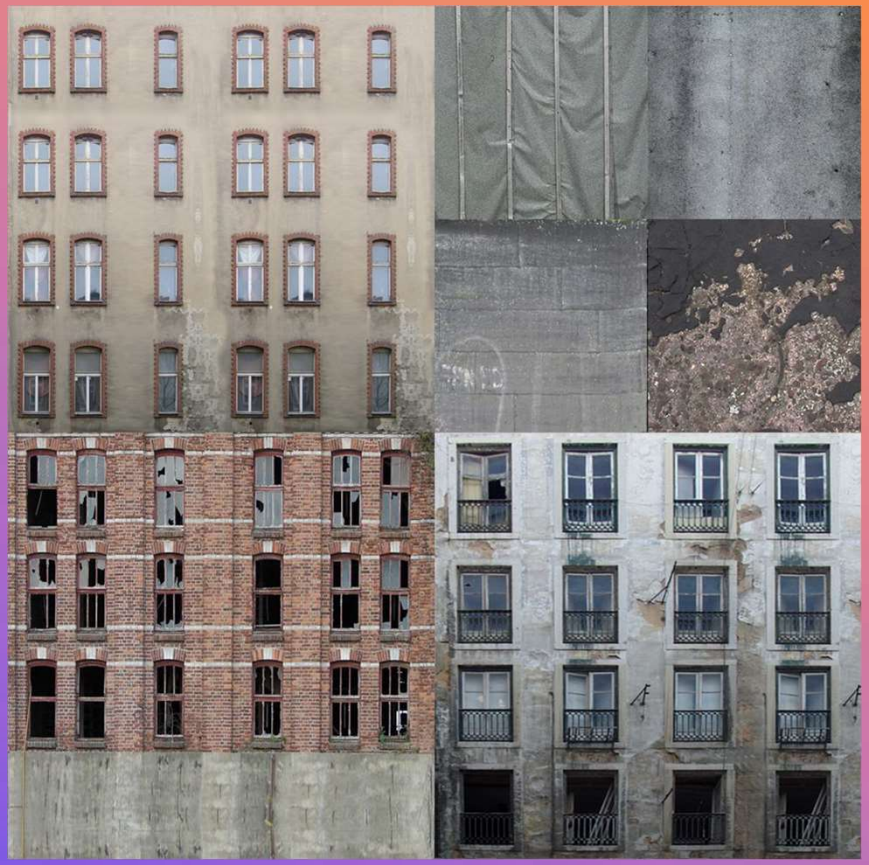
Ram / Disk ↑



Texture Atlas / Spritesheet

CPU ↓↓↓

Ram / Disk ↓



ลด Quality ของเกม

CPU / GPU ↓↓↓

Ram / Disk ↓↓↓

- ปรับ Quality Setting
- ลด Resolution

ปัญหา Memory

Texture Asset Tips

- Texture สามารถ Tint ได้
- ใช้ขนาด Texture ให้เหมาะสม
- การเลือกใช้ Compression
- การ Scale Texture ที่มีสีเดียว , หรือ Gradient
- 9 Sliced Texture

Audio Asset Tips

Load Type

- Decompress on Load : Decompress แล้วเก็บไว้ใน Ram , เสีย Ram เยอะ แต่ไม่เสีย CPU ตอนเล่น
- Compressed in Memory : บีบอัดไฟล์เก็บไว้ใน Ram , ใช้ CPU ตอนเล่นเสียง
- Streaming : เก็บไฟล์ไว้ใน HDD , Decode เสียงขณะเล่นเพลง

Compression Format

- PCM : Quality ดีที่สุด แต่ขนาดไฟล์ใหญ่ --> ใช้กับไฟล์ SFX เล็กๆ
- ADPCM : Quality ปานกลาง ขนาดไฟล์ปานกลาง --> ใช้กับ SFX เล็กๆที่มี Noise
- Vorbis : Quality ต่ำสุด ขนาดไฟล์เล็ก --> ใช้กับเพลงยาวๆ หรือ SFX ขนาด

Audio Asset Tips (ต่อ)

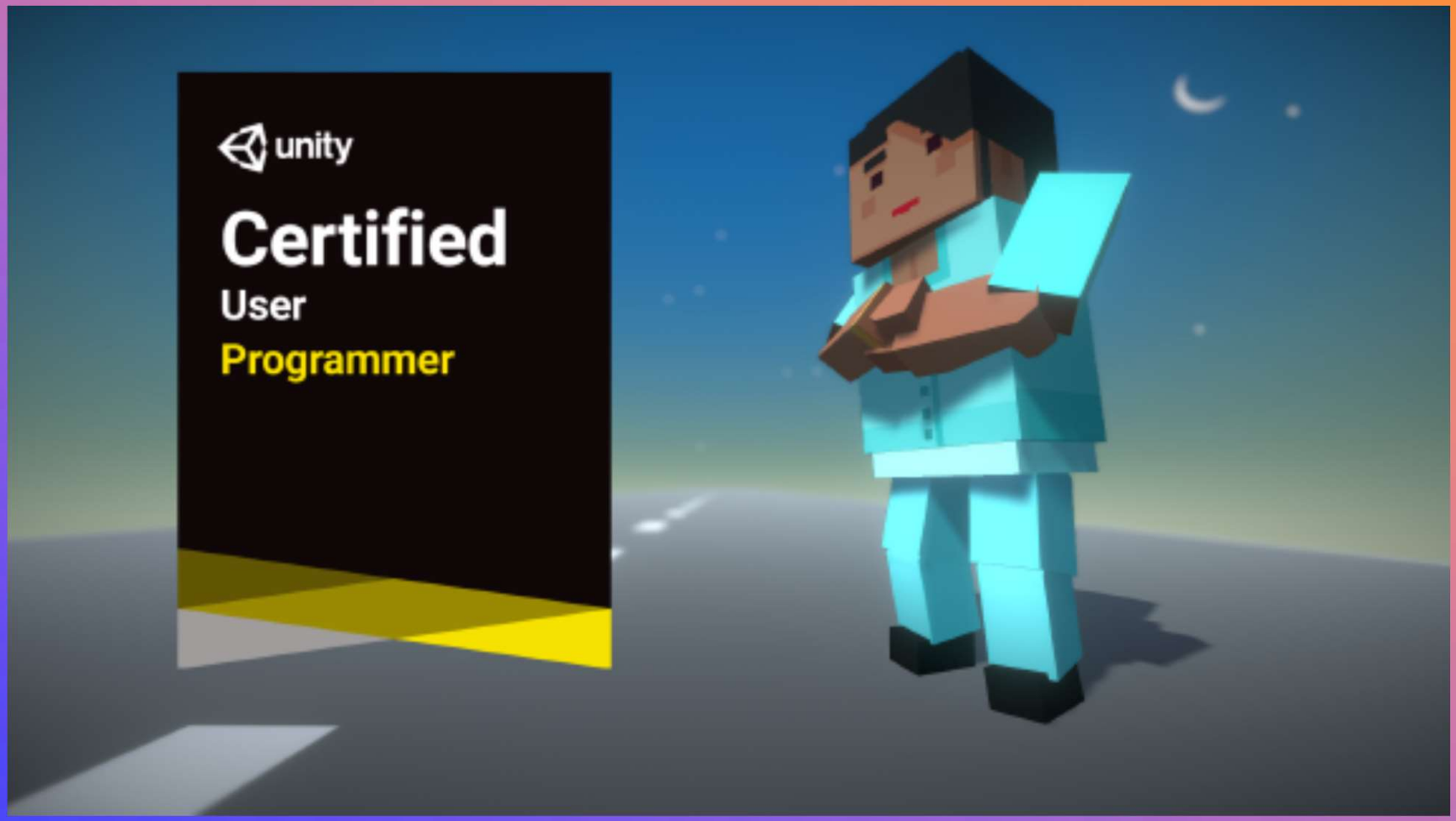
Music

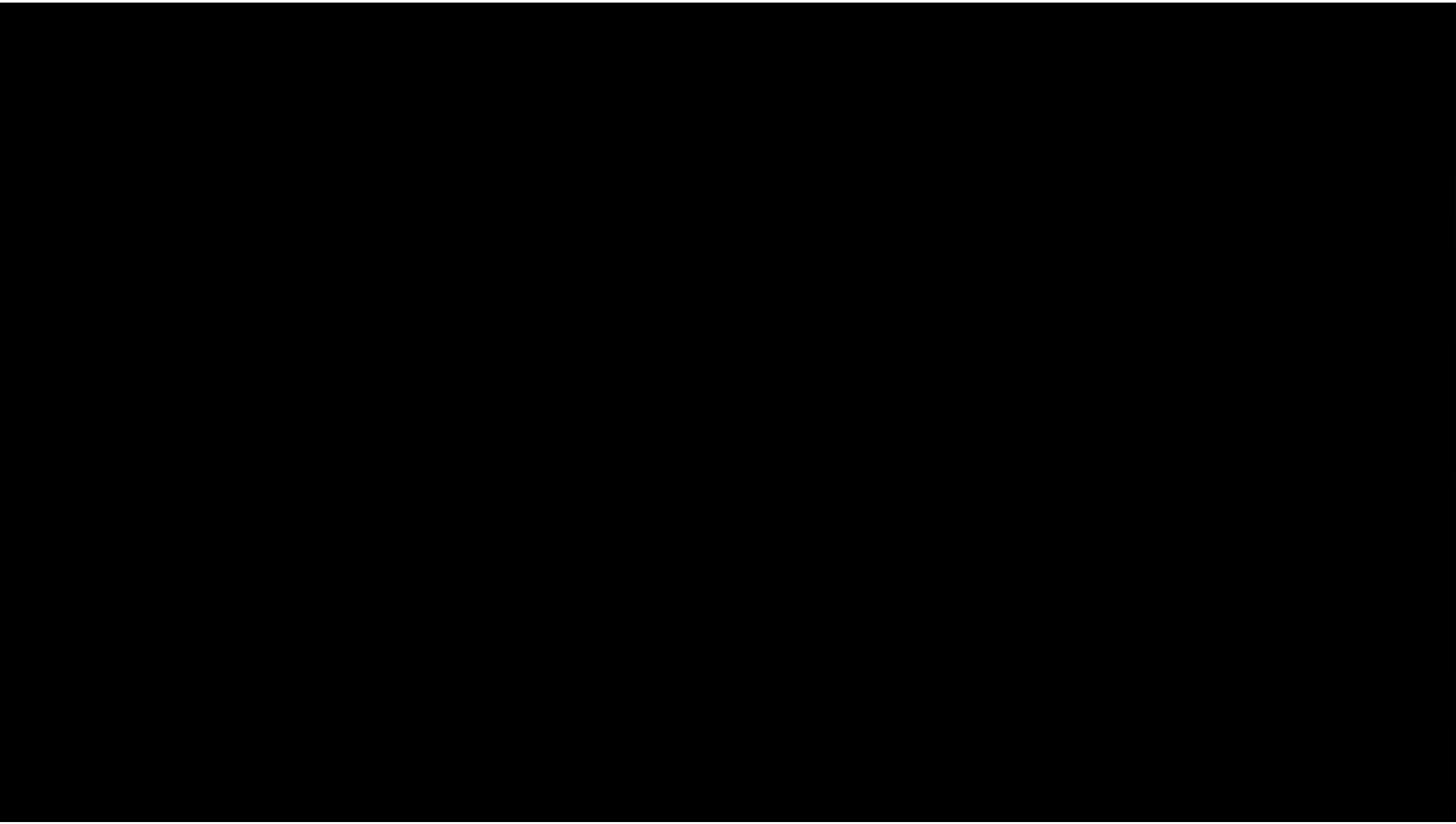
- Streaming + Vorbis
- Compressed + Vorbis

SFX

- ใช้งานบ่อย , เสียงสั้นๆ : Decompress + PCM / ADPCM
- ใช้งานบ่อย , เสียงยาวปานกลาง : Compressed + ADPCM
- ใช้งานไม่บ่อย , เสียงสั้นๆ : Compressed + ADPCM
- ใช้งานไม่บ่อย , เสียงยาวปานกลาง : Compressed + Vorbis

Reference : http://blog.theknightsofunity.com/wrong-import-settings-killing-unity-game-part-2/


unity
Certified
User
Programmer



Project Optimization

Techniques:

1: Variable attributes

2: Unity Event Functions

3: Object Pooling

1: Variable attributes

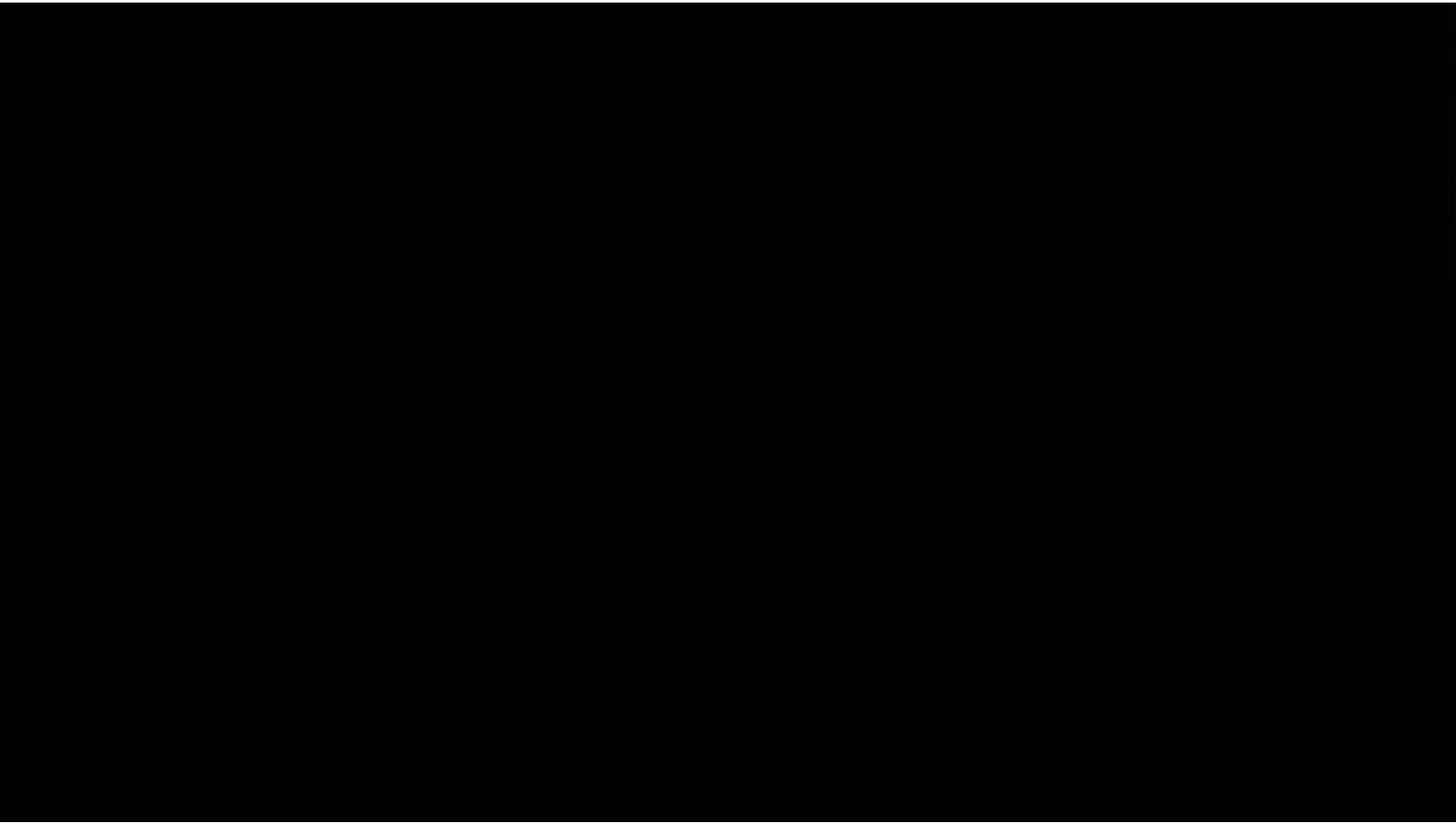
In the course, we only ever used "public" or "private" variables, but there are a lot of other variable attributes you should be familiar with.

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

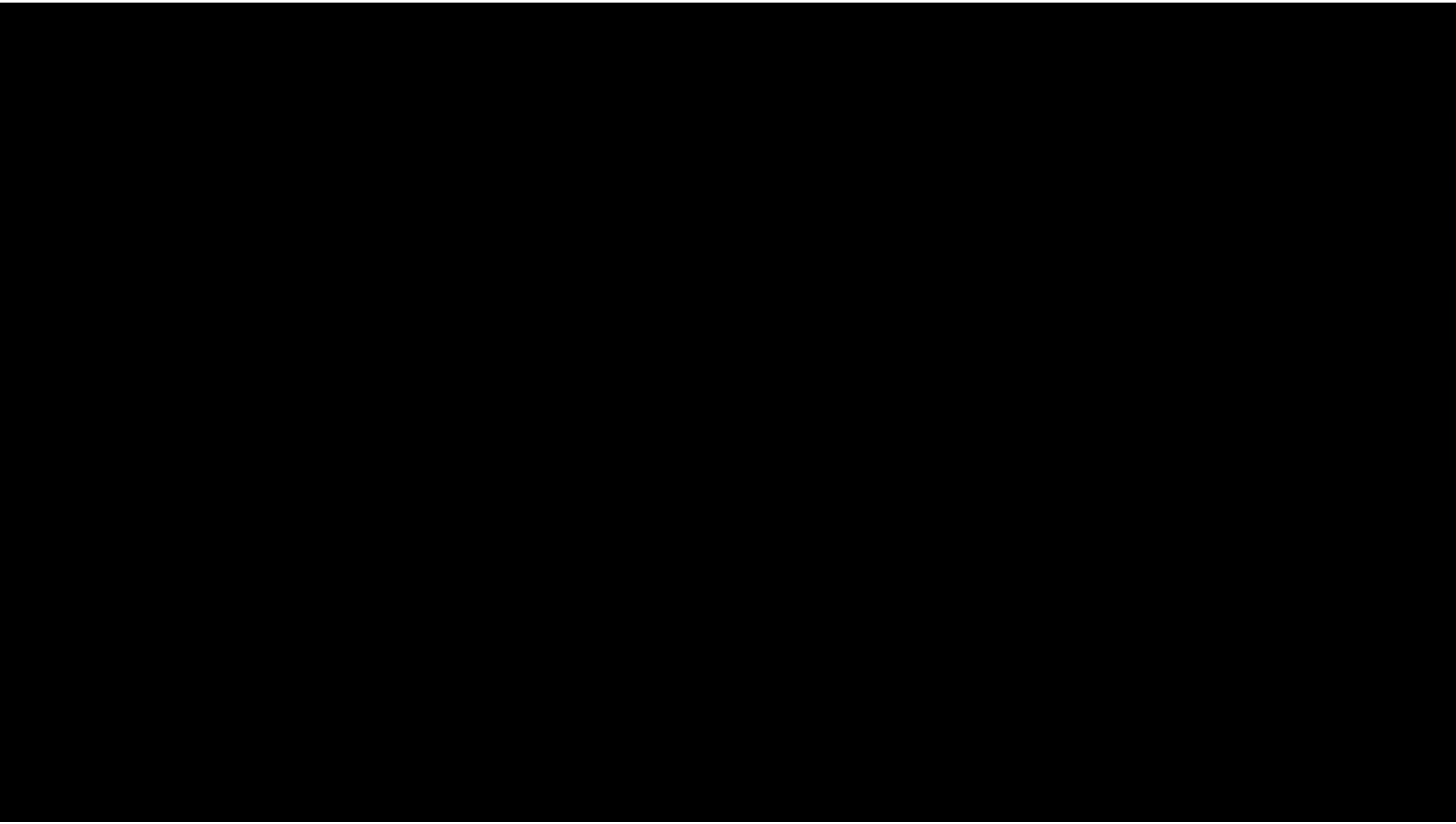
In the course we only ever used the default Update() and Start() event functions, but there are others you might want to use in different circumstances.

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() { ...
```

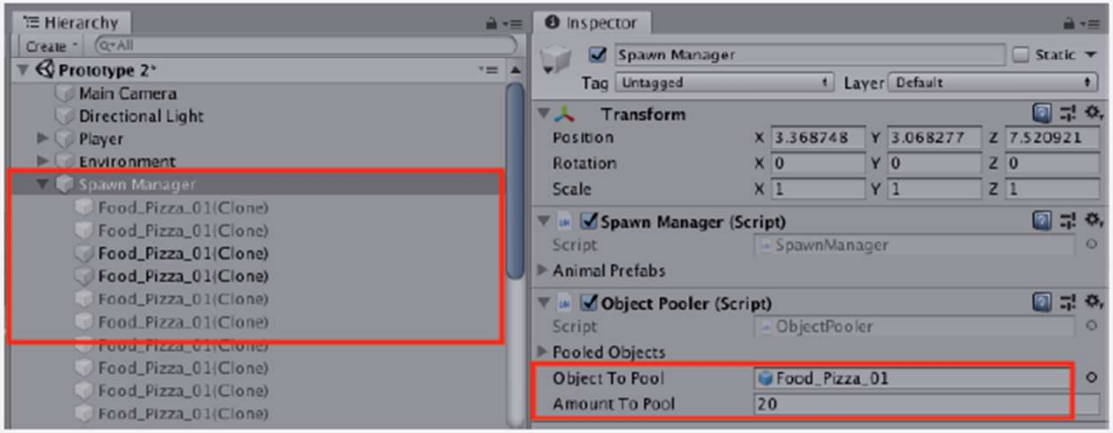


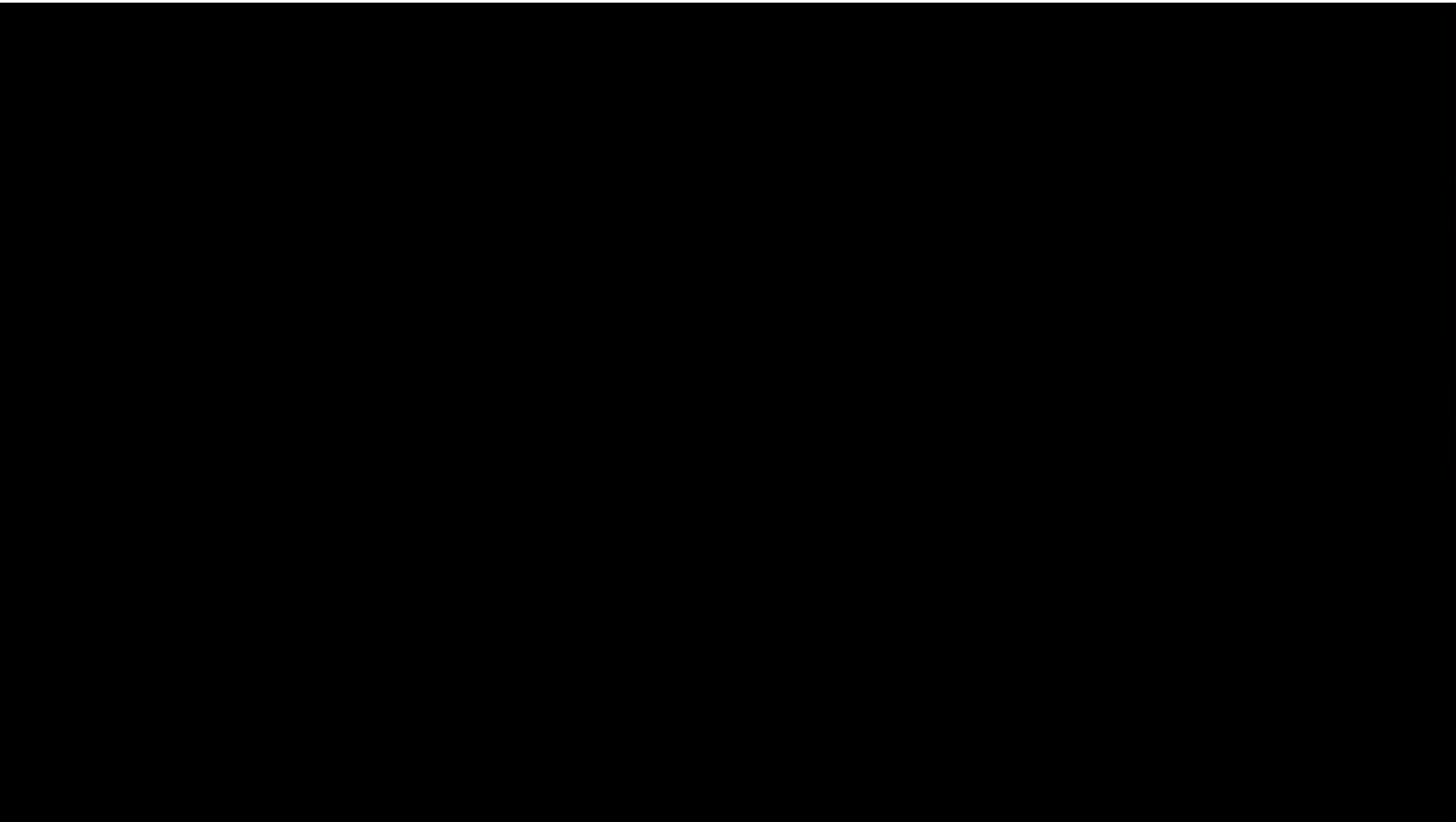
3: Object Pooling

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





Troubleshooting

Steps:

Step 1: Make the vehicle use forces

Step 2: Prevent car from flipping over Step

Step 3: Add a speedometer display Step

Step 4: Add an RPM display Step

Step 5: Prevent driving in mid-air

Step 1: Make the vehicle use forces

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.

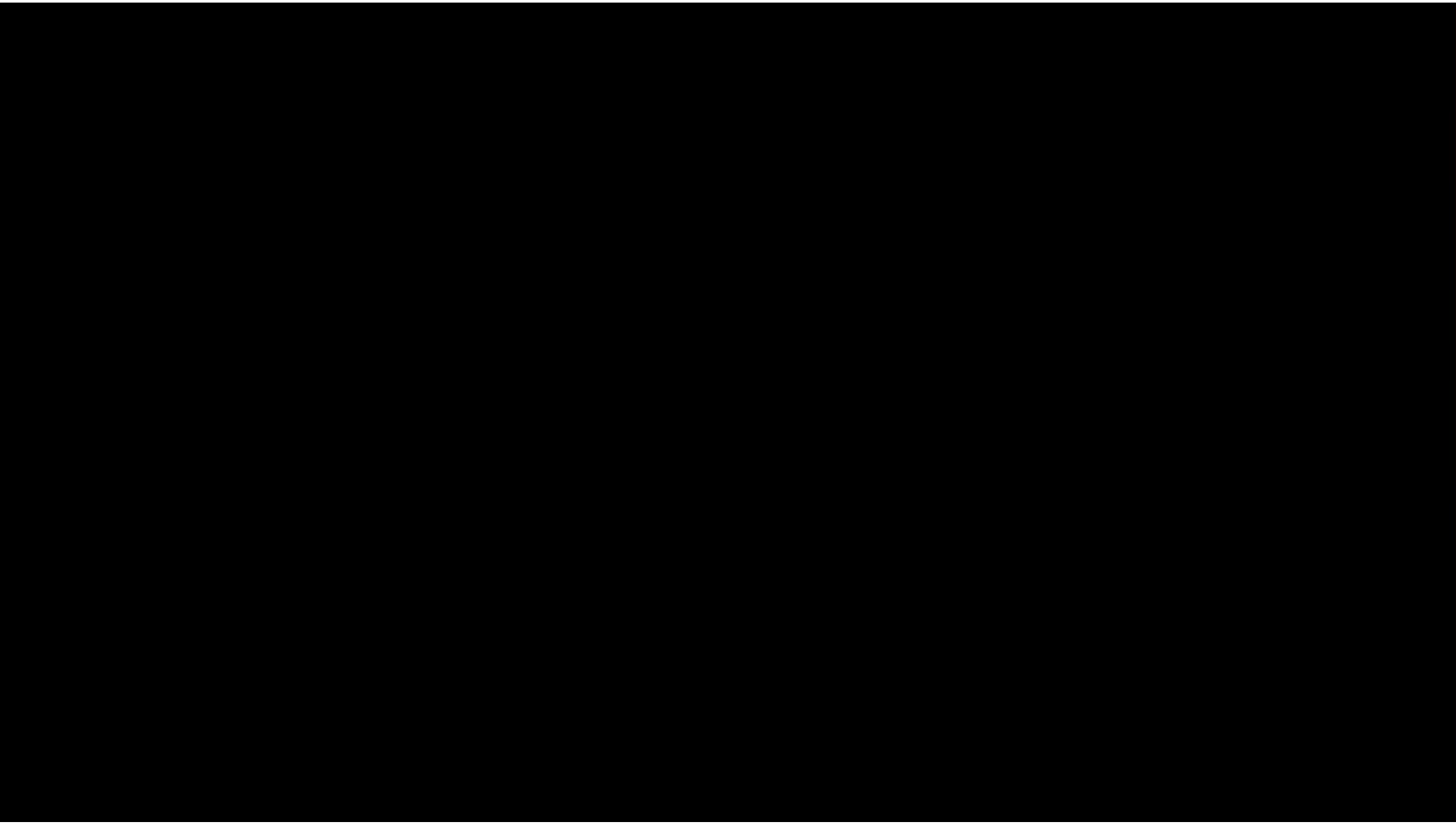
1. Open your **Prototype 1** project and make a backup
2. Replace the Translate call with an AddForce call on the vehicle's Rigidbody, renaming the "speed" variable to "horsePower"
3. Increase the ***horsePower*** to be able to actually move the vehicle
4. To make the vehicle move in the appropriate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

```
[SerializeField] 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

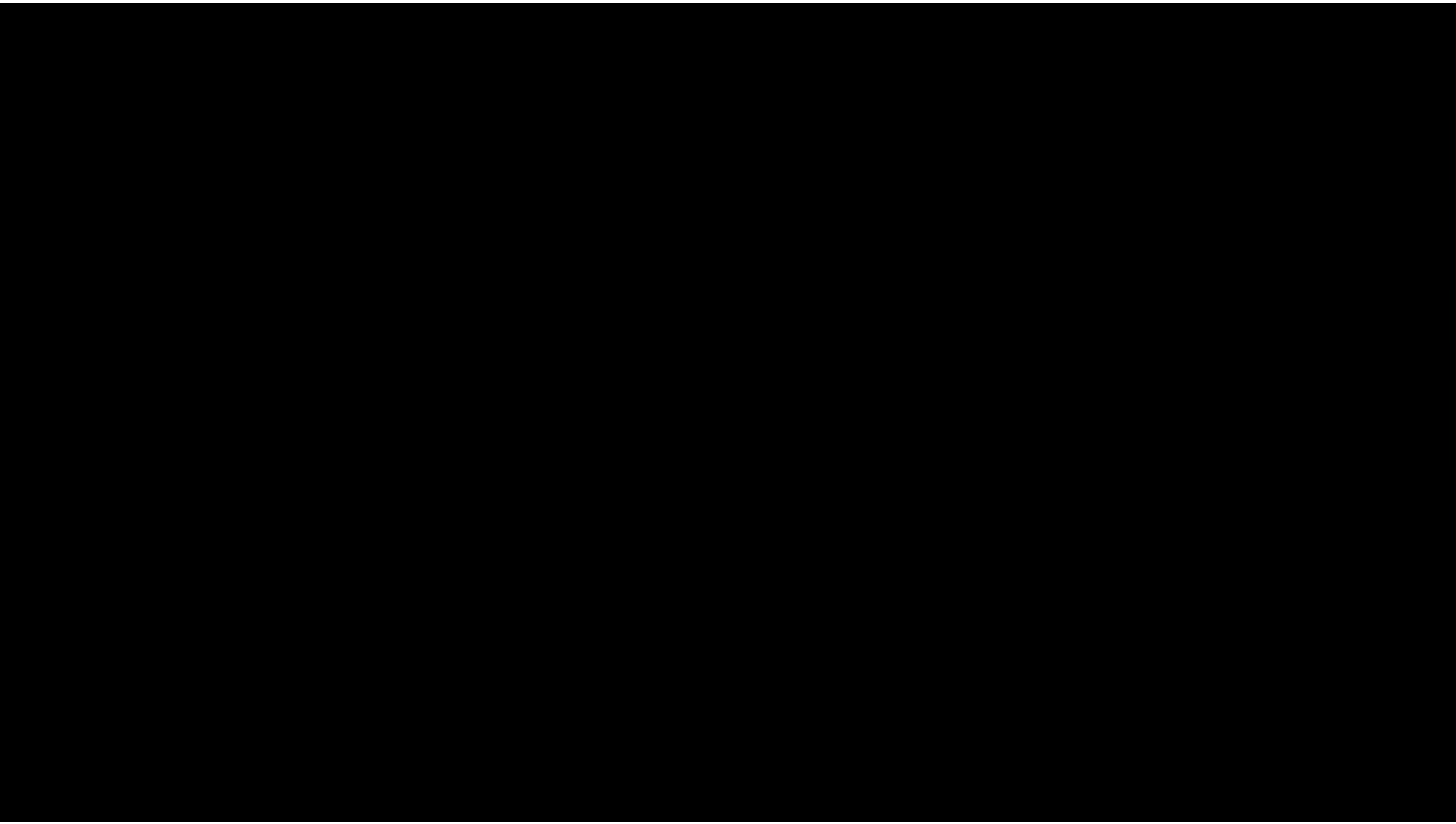
Now that we've implemented real physics on the vehicles, it is very easy to overturn. We need to figure out a way to make our vehicle safer to drive.

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.

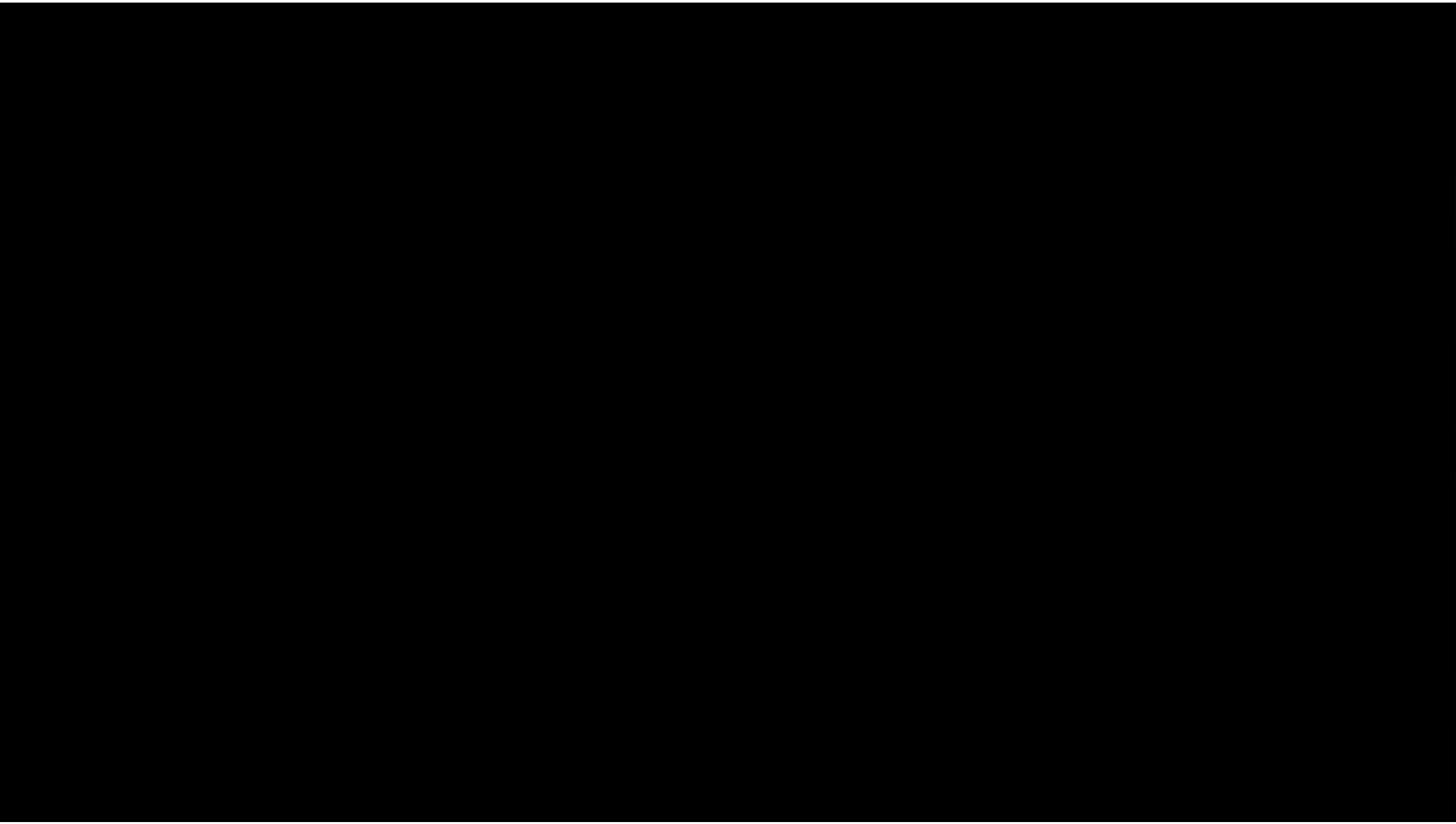
1. Add a new ***TextMeshPro - Text*** object for your "Speedometer Text"
2. Import the **TMPro library**, then create and assign new create a new **TextMeshProUGUI** variable for your ***speedometerText***
3. Create a new float variables for your ***speed***
4. In Update(), **calculate** the speed in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

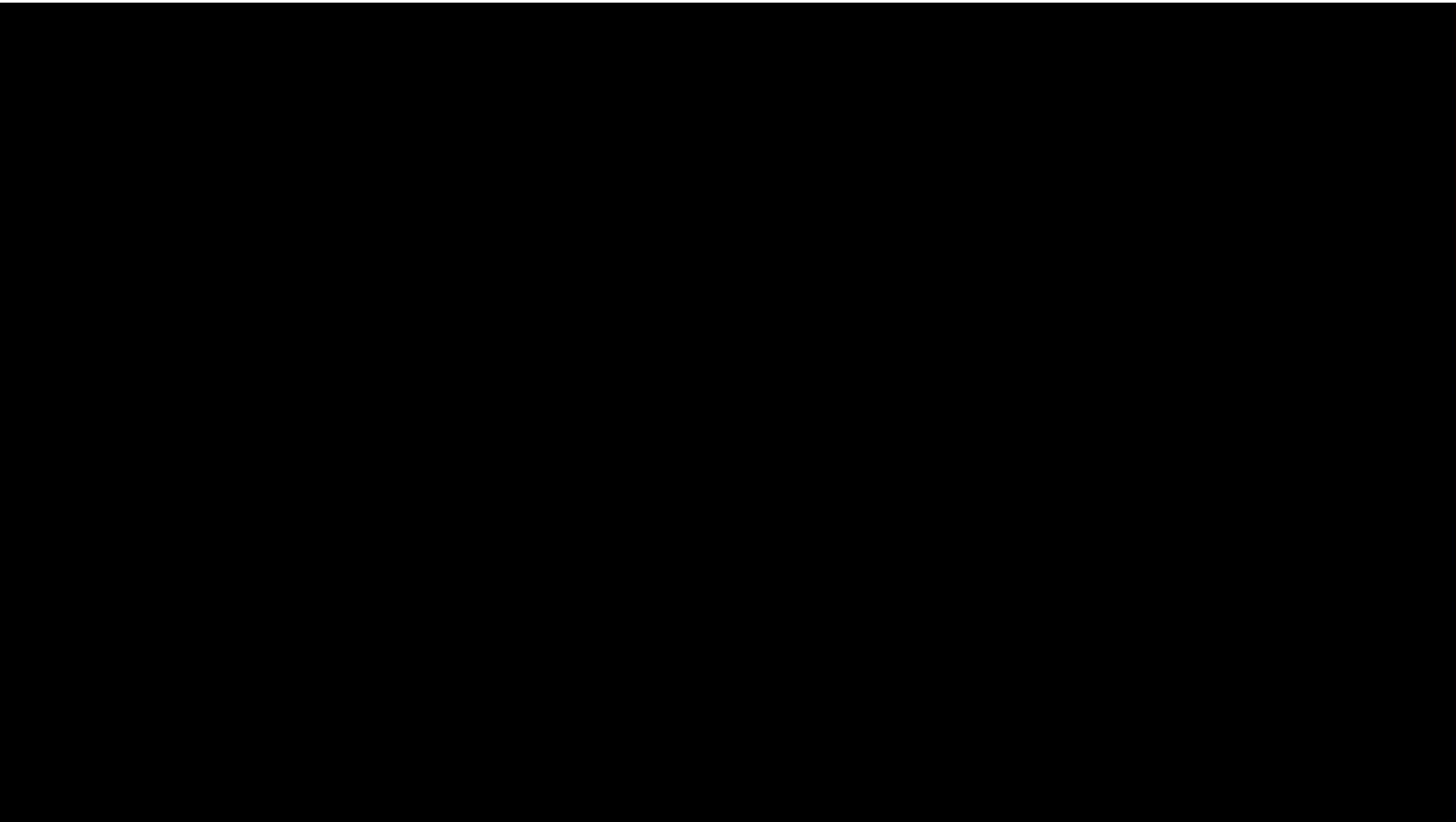
One other cool feature that a lot of car simulators have is a display of the RPM (Revolutions per Minute) - the tricky part is figuring out how to calculate it.

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!

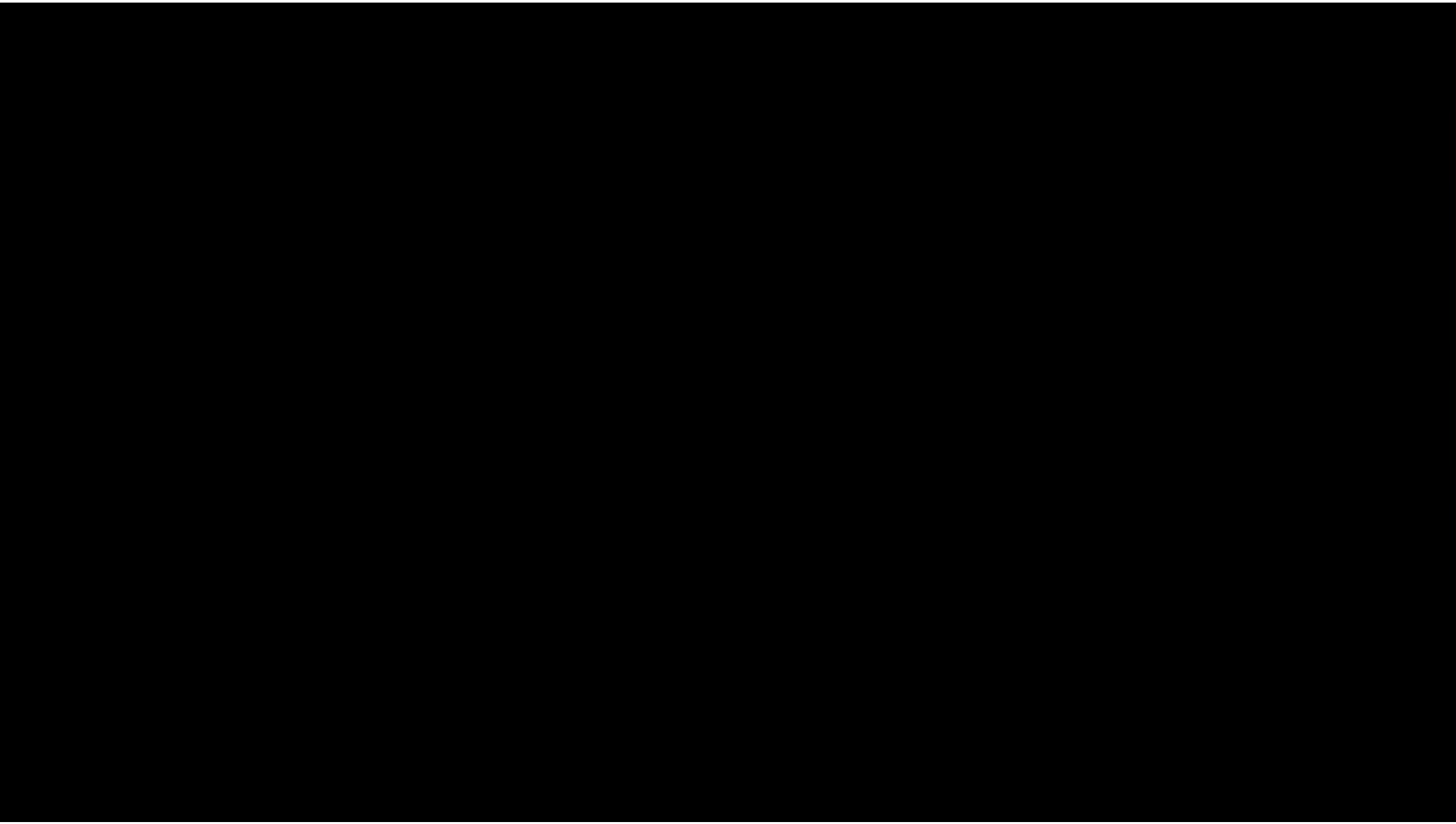
1. Declare a new **List** of **WheelColliders** named ***allWheels*** (or frontWheels/backWheels), then assign each of your wheels to that list in the inspector
2. Declare a new ***int wheelsOnGround***
3. Write a ***bool IsOnGround()*** method that returns true if all wheels are on the ground and false 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 == 4) {
    return true;
  } else {
    return false;
  }
}
```



Sharing your Projects

Steps:

Step 1: Install export Modules

Step 2: Build your game for Mac or

Step 3: Build your game for WebGL

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 navigate to the **Installs** Tab
2. On the Unity version you've been using in the course, select **Add Modules**.
3. Select **WebGL Build Support**, and either **Mac** or **Windows** build support, then click **Done** and wait for the installation to complete

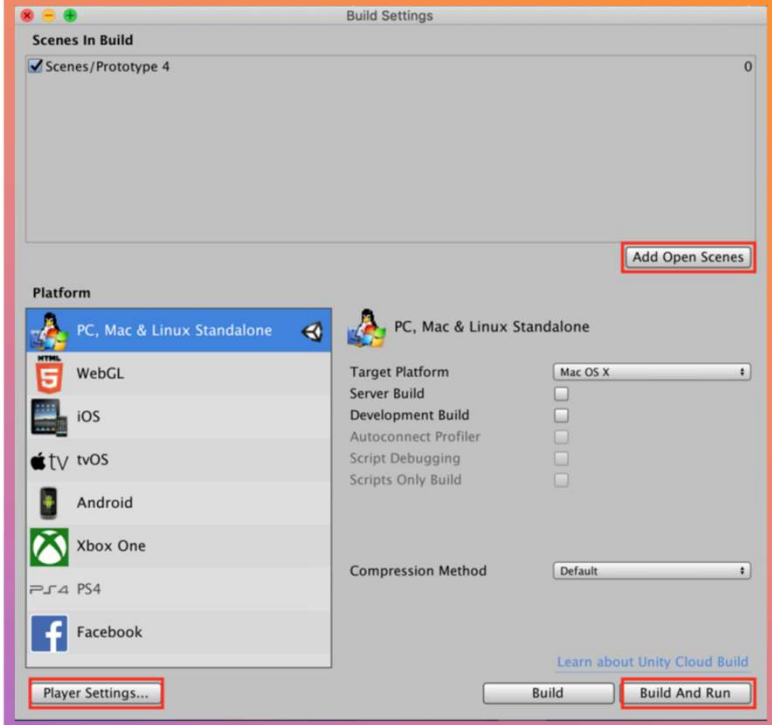
- **Tip** - Mac and Windows will create apps for your computer and WebGL will allow you to publish online
- **Tip** - you should see little icons appear when it is complete
- **Tip** - WebGL is nice because you can more easily share it online and it is platform-independent

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".
   For more information, check out the documentation on configuring player settings.
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

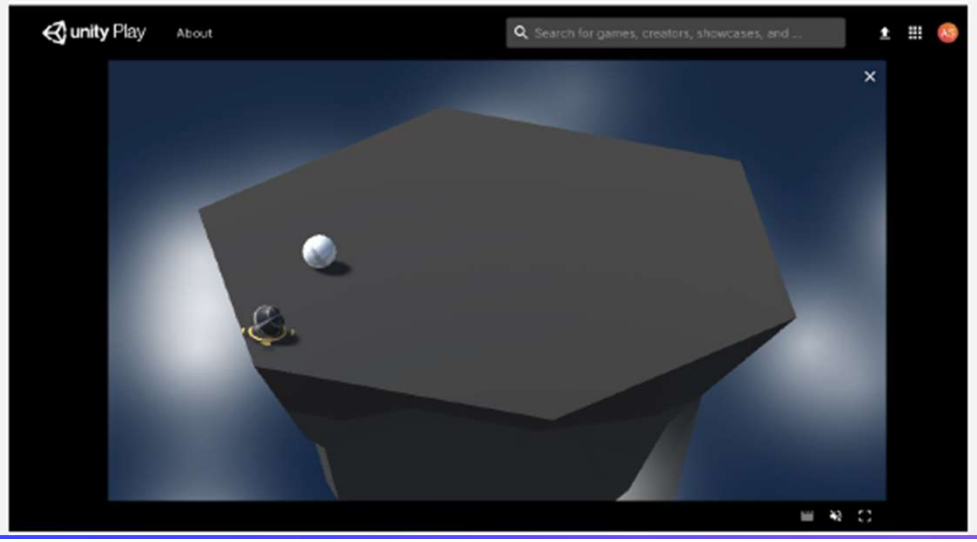


Step 3: Build your game for WebGL

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.**
   Note: you will only be able to do this if you have installed the WebGL Build Support export module
2. Click **Build**, then save in your "Builds" folder with " - WebGL" 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 Unity Play or 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"



THANK YOU