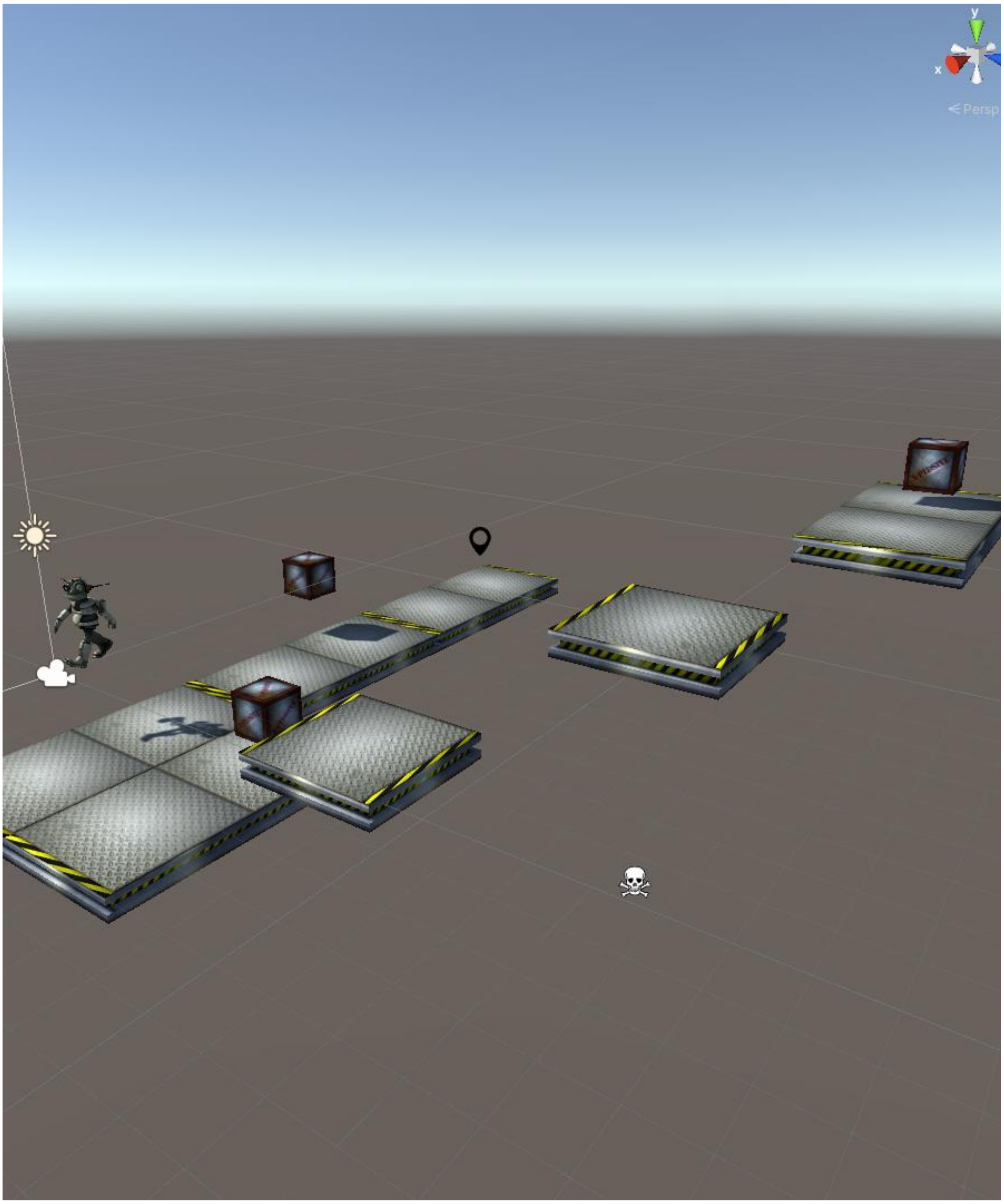
Professur für Erweiterte Realität
Fakultät für Informatik
Technische Universität München

Tutorial & Exercises – Tutorial 8

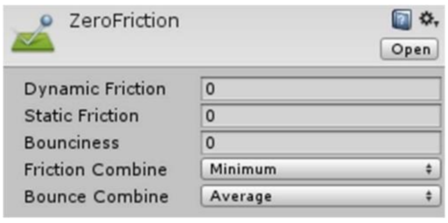
Platformer



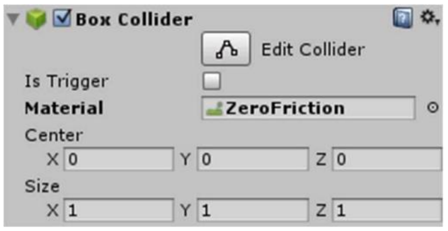
Lets do more for your game (Prepare before class)

Physic Materials

When you run into an obstacle (a GameObject with a Collider attached), you will notice that your Player flies upwards. To prevent this, create a Folder "Physic Materials" and add a new Physic Material (Create Physic Material), and name it "ZeroFriction". You can apply it to obstacles where the player shouldn't bounce.


ZeroFriction

Dynamic Friction	0
Static Friction	0
Bounciness	0
Friction Combine	Minimum
Bounce Combine	Average


Box Collider

Is Trigger	☐
Material	ZeroFriction
Center	X 0 Y 0 Z 0
Size	X 1 Y 1 Z 1

Physic Materials are used to adjust friction and bouncing effect of colliding GameObjects.

Set friction and bounciness to zero, and make sure the minimum friction (zero) is used when a collision occurs. Apply it to your obstacles by dragging it onto the Collider's Material.

Careful: If used on non-static objects like crates, they will slide without friction. To prevent your player from bouncing off of crates, increase the player's mass or reduce the mass of the crates.

Try different Physic Materials for different GameObjects.

Singletons: Handling Game Data

A singleton is a design pattern that restricts the Instantiation of a class to one object. This is a useful approach to store information like the player's score and lives.

- Create a new C# script named "GameData".
- Delete "using…". Delete ": MonoBehaviour" to make sure, the singleton script does not derive from MonoBehaviour.
- At first make sure there is only one instance of GameData:

```
private static GameData instance;
private GameData()
{
    if (instance != null)
        return;
    instance = this;
}
public static GameData Instance
{
    get
    {
        if (instance == null)
        {
            instance = new GameData();
        }
        return instance;
    }
}
```

- Now add a private variable to represent the player’s score. Can be accessed with get and set.
- Try another approach for simple variables to get and set game information; use the following expression, where “Lives” is a variable itself.

This script does not need to be on a GameObject. You can simp-ly access the score by typing GameData.Instance.Score.

```
private int score = 0;
public int Score
{
    get
    {
        return score;
    }
    set
    {
        score = value;
    }
}
```

```
public int Lives
{
    get;
    set;
}
```

Tutorial (do within class)

Adding a Moving Enemy

Drag the “Bomb Red” Prefab from the PowerUps Package into your scene and rename it to BombEnemy. Add Components for movement and collisions:

- Add a Sphere Collider and change its radius as you like. (Example Values: Center = (0.022, -
- 0.024, 0); Radius = 0.1)
- Add a Rigidbody. Open Constraints and select “Freeze Z position”.
- Set the GameObject’s x rotation to 300 (it looks better).

We will create a script “Enemy” in which we will handle the enemy’s movement.

- Create Enemy.cs, attach it to the enemy GameObjct and open it.
- Create a new public float variable speed.
- Create a new variable enemyRigidbody of the type Rigidbody.
- Delete the Start and Update functions, because we will – as you already know – handle the Rigidbody movement in FixedUpdate(). Add the FixedUpdate function.
- In Awake() assign the GameObject’s Rigidbody to your enemyRigidbody variable (just as you did in PlayerMovement.cs). To make sure, that the GameObject has a RigidBody Component, we use RequireComponent for our class.

```
[RequireComponent(typeof(Rigidbody))]
public class Enemy : MonoBehaviour {
    public float speed;
    Rigidbody enemyRigidbody;
    void Awake()
    {
        enemyRigidbody = gameObject.GetComponent<Rigidbody>();
}

void FixedUpdate()
{
    enemyRigidbody.velocity = new Vector3(speed, enemyRigidbody.velocity.y, 0);
}
```

- Add code that manipulates the Rigidbody’s velocity so that it moves on a straight line: The x velocity will be speed, y will not be changed, and z will be 0.

Test the code! Change the speed as you like. Also save your BombEnemy in a new Prefab for repeated use.

The Enemy will now simply go on and fall down the platform when it reaches its end. We need to add collisions with a trigger where we want the Enemy to change its direction. Create an Empty GameObject, and give them the tag “End”. Add a Box Collider, and mark it as trigger. Rename the Object to “TurnTrigger”, and save it as a Prefab. Place it at the end of the platforms where the Enemy should turn.

Now you need to expand the PlayerBehaviour script.

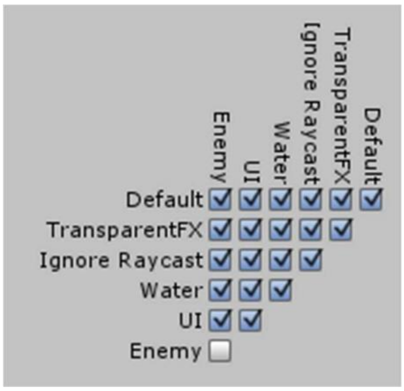
- Check if the tag of the GameObject the enemy is colliding with is “End”.
- Change its moving direction changing the sign of speed.

```
void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag ==
    "End")
    {
        speed *= -1;
    }
}
```

Removing Collisions between the Enemies

All your enemies will collide with each other, but we want to avoid this. Therefore we create a new Layer and disable collisions for all GameObjects within this Layer.

- Select the enemy and add a Layer “Enemy”. Assign it to the enemy.
- Go to Edit □ Project Settings □ Physics
- Deselect the Enemy/Enemy field in the Layer Collision Matrix.



	Enemy	UI	Water	Ignore Raycast	TransparentFX	Default
Default	✓	✓	✓	✓	✓	✓
TransparentFX	✓	✓	✓	✓	✓	
Ignore Raycast	✓	✓	✓	✓		
Water	✓	✓	✓			
UI	✓	✓				
Enemy	☐					

Player Collision

Add collision with the Player and the Enemy: When jumping on an Enemy, the Enemy should die, otherwise the Player dies. You can add invincible enemies, which always kill the player on collision.

To identify Enemy add a tag “Enemy”.

Let us expand Enemy.cs at first.

- Add a public function OnDeath() in which you disable the collider. This will make the enemy fall down and it will continue falling down unless you handle it for example by destroying it

```
public void OnDeath()
{
    gameOb-
    ject.GetComponent<Collider>
    ().enabled = false;
}
```

when it enters the Death Zone. However, we will refrain from doing this because we later want to implement a time reversal mechanics and we will need the Enemy again.

- Add a boolean with which you can determine whether an enemy is invincible, and a float in which we will save the speed with which the player will bounce off.

```
public bool invincible;
public float bumpSpeed;
```

- In the Inspector set bumpSpeed to 20.

Now, we want to adjust PlayerBehaviour.cs.

- At first we will create a new function void OnDeath(), which is called when the player dies. For now, this function will only call Spawn();

```
void OnDeath()
{
    Spawn();
}
```

How the player reacts to colliding with the enemy, depends on different factors: Is the enemy invincible? Has the player jumped onto the enemy?

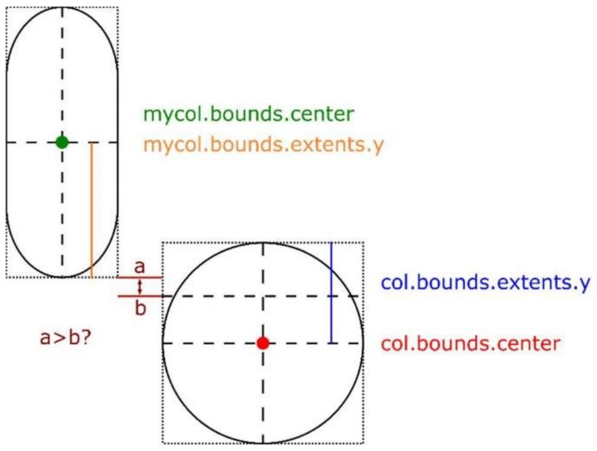
- Add OnCollisionEnter(Collision other) and at first check if the Collision’s GameObject’s tag is “Enemy”:

```
void OnCollisionEnter(Collision other)
{
    if (other.gameObject.tag == "Enemy")
    {
        Enemy enemy = other.gameObject.GetComponent<Enemy>();
        Collider col = other.gameObject.GetComponent<Collider>();
        Collider mycol = this.gameObject.GetComponent<Collider>();
if (enemy.invincible)
        {
            OnDeath();
        }
    }
}
```

- Then we want to check whether the player has jumped onto the enemy. If so, the player should bump off and the enemy should die. We use the collider bounds (= axis aligned bounding box) of the player and the enemy to determine if the player is (reasonably) above the enemy at the time of the collision

- If this is the case, we will call a new function JumpedOnEnemy(float bumpSpeed), which takes the enemy’s bumpSpeed as a parameter. Also, we will call the enemy’s OnDeath function.

```
else if (mycol.bounds.center.y - mycol.bounds.extents.y >
    col.bounds.center.y + 0.5f * col.bounds.extents.y)
{
    JumpedOnEnemy(enemy.bumpSpeed);
    enemy.OnDeath();
}
```



- Now add JumpedOnEnemy, in which we will apply a force to the Rigidbody to simulate the bouncing effect.

```
void JumpedOnEnemy(float bumpSpeed)
{
    playerRigidbody.velocity  = new Vec-
    tor3(playerRigidbody.velocity.x, bumpSpeed, playerRigid-
    body.velocity.z);
}
```

In all other cases, we will call OnDeath().

```
else
{
    OnDeath();
}
```

If everything works, create a new folder "Prefabs” and save the Enemy as a Prefab.

Accessing Game Data

We will increase the score when the player has jumped on an enemy.

Go to PlayerBehaviour.cs and add in OnCollisionEnter right before you call JumpedOnEnemy:

```
GameData.Instance.Score += 10;
```

If you want different enemy types to give different points, add a new variable in Enemy.cs and increase score by this value.

Do the same for you collectibles!

Checkpoints

As your Player can die now, add Checkpoints to your level. If your character reaches a Checkpoint, he will respawn at this position when dying afterwards.

Homework (do after class)

Exercise: Collectibles and Further Improvements

The final goal of the tutorial is to build a complete 3D Platformer Level. You should now have a working character which you can control, and you should have obstacles and enemies. Make your game more interesting by adding collectibles and giving your level a different

Exercise 1: Add Collectibles

Add some collectibles to your level. Use one of Unity's 3D Objects or an own 3D model. You can also download the powerups.unitypackage from moodle or find it in Unity's asset store. To use the package from moodle, click on Assets · Import Package · Custom Package.

For this exercise it is enough, that the collectibles are destroyed when your player character is running into them.

Optional: Add rotations, particle effect, etc. as you like. To get some inspiration you can look at popular platformers. Also, think about how you want your game to look in the end. Maybe you already find collectibles that support the intended style and mood.

Exercise 2: Add a Unique Appearance

You have already worked some time on your platformer game, so it is time to think about its look and feel. How should your game look like and what should its atmosphere be?

- Assign Materials to your GameObjects with matched textures, especially to your platforms and obstacles. Use at least two different Materials. Also, try using normal maps (or other main maps).
  If you need assistance, watch Unity's tutorial video about the built in Standard Shaders and explore their possibilities.

- Add a skybox that supports the atmosphere of your game. You can download and import an asset package with skyboxes from the asset store (for example this or this skybox).
- Skyboxes are applied via Window -> Lighting. Select a skybox material (Environmental Lighting).

Exercise 3: Graphical User Interface

Your character already has lives and can gain points, and now we want to display them. Create a new UI Text GameObject in order to display the player’s live and score. Create a new script that handles to always update the shown text.

- In the spaceshooter tutorial you already learnt how to do this.

Technische Universität München
Fakultät für Informatik
Professur für Erweiterte Realität

Boltzmannstr. 3
85748 Garching bei München

https://far.in.tum.de