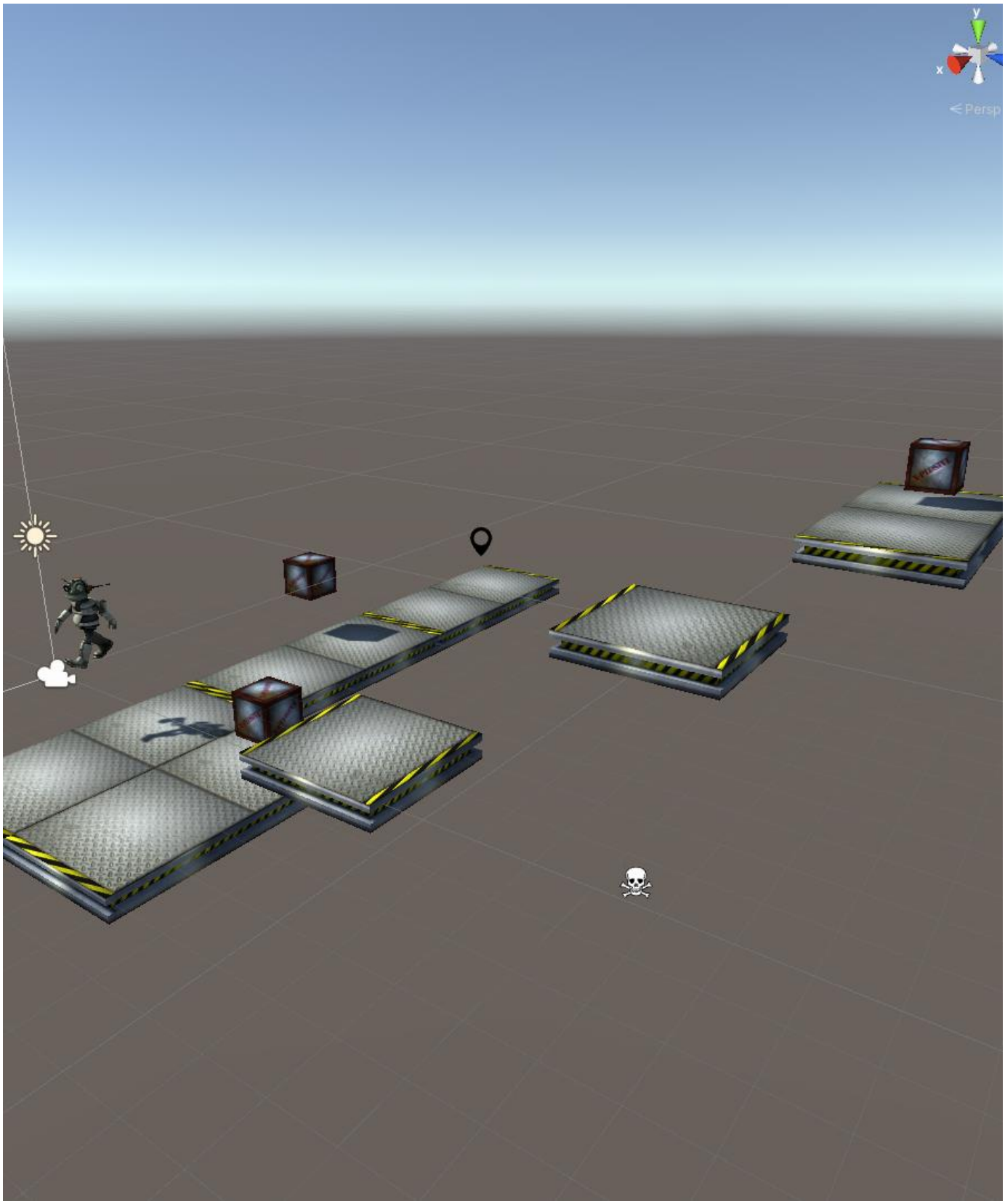
Professur für Erweiterte Realität
Fakultät für Informatik
Technische Universität München

Tutorial & Exercises – Tutorial 6

Platformer



Prepare your scene

Unity helps you create 2D, 2.5D and 3D games. The space shooter you created in the first half of this exercise followed the rules of 2.5D and thus had restricted movement. Now we will explore the full possibilities of a 3D world and 3D gameplay.

The overall goal is to build a complete level of a 3D platformer.

Setting the Scene

Tip: When setting the scene, you can remember which colour correlates to which axis with the simple mnemonic device “RGB = XYZ”.

Building a Basic Platform Structure

We will start with building a basic platform structure, which you will extend later.

- You can use Unity’s 3D GameObjects (like planes, cubes and cylinders), or look for or create your own 3D models.
  For example: Create a new cube and scale it to (10, 1, 10). Place a smaller cube (10, 1, 5) next to it. Leave a little gap between them, that the player needs to jump over.
- Use prefabs! This will make it easier to change properties later.
- Take advantage of the 3D space and be creative!
- Use tiling.

If you want to fit GameObjects seamlessly, try vertex snapping: press and hold the V-key when positioning GameObjects. It lets you take any vertex from a given mesh and with your mouse place that vertex in the same position as any vertex from any other mesh you choose.

Work with whole unit length (e.g. length of 1) to easily set the platform seamlessly.
Then you only need to enter integer values in the Inspector.

Tiling

You can build your platforms out of smaller, tileable pieces. The GameObjects then will fit together seamless. You can also tile textures: One texture can be used multiple times on a single GameObject.

Tiling was a very common technique in the early days of computer games. It had the very useful advantage of reducing the amount of graphical assets required, keeping the project’s overall size down. Today, tiling remains a useful technique as it reduces both asset production time and download size.

Professur für Erweiterte Realität
Fakultät für Informatik
Technische Universität München

Exercises (Prepare before class)

Building a small level
The final goal of the tutorial is to build a complete 3D Platformer Level. Further enhance your level with the new Platforms from **Homework 1.**

Add moveable Platforms
To introduce challenging obstacles for your new game think about small platforms with a specific behavior**.**

For the start we want to have a platform moving between to specific positions

Tutorial (do within class)

Adding a Player Character

Figure 1:norT © S. Liedtke, P. Tolstoi D. Giebert

Rigidbody Movement: Making the Player Character Jump and Run

As the basics are set, you are ready to add a player character.

In the space shooter you moved your ship by translating the Transform component. Now you will learn about another way of moving a GameObject using its Rigidbody component.

- Create a new Capsule that will be your player character. Name it “Player”, and position and scale it if necessary.
- Add a Rigidbody to the Capsule.
- Make sure Gravity is checked and Is Kinematic is not checked. We want the physics engine to apply gravity to our Rigidbody and use the gravity to jump.
- Select “Interpolate” for the property Interpolate. This will smooth out the effect of running physics at a fixed step. Graphics are rendered at variable frame rate, so physics and graphics are not completely in sync. This can lead to jittery looking objects, and is often visible on the player character, especially if a camera follows it. Turn Interpolate on for your Player GameObject, but not for any other GameObject.

By default, we will control the movement with WASD and have the mouse position dictate which way the character is facing. But we will design our script in a way, that this can be changed, and you can determine the movement axes yourself. Also, your Character will be able to jump.

- Create a C# script “PlayerBehaviour”.
- Delete the Start and the Update function (it can be easier to structure your script, if it is empty).

Functions:
As the character should be able to run (both forward and sideways), to turn and to jump we need to add corresponding functions. We also need a function with which we can save the user input. All these functions must be called either in Update() or in FixedUpdate(), depending on whether they use physics or not. Only Run() and Jump() will use the GameObject's Rigidbody, and thus physics. In Awake() we want to set the start values.

To check if the character is standing on the ground, we will need a function that returns true if the character is grounded. (You can for example place it before your Awake function.)

- bool Grounded(): check if the player is currently grounded.

Helper Classes and Public Properties:
We need different public variables that can be broadly categorised into variables concerning movement settings and input settings. Therefore, we will add two new classes ***in*** our class PlayerBehaviour.

- Add two new classes within the existing class in order to structure the variables:

- Declare these classes as "Serializable" by adding System.Serializable in the line above each class.

- The Serializable attribute lets you embed the class with sub properties in the inspector.

Now we will add different properties to the classes, beginning with MoveSettings:

- The movement settings cover attributes regarding the different movement velocities (run, rotate and jump). Also we want to check if the GameObject is grounded. This is why we need a Layermask to mark the ground, and a float in which we store the GameObject's distance to the floor. Is the distance between the GameObject and the floor bigger than this value, it is not grounded.

```
public float runVelocity = 12;
public float rotateVelocity = 100;
public float jumpVelocity = 8;
public float distanceToGround = 1.3f;
public LayerMask ground;
```

- LayerMasks are useful to filter GameObjects when using Raycasts. They refer to the *Layer* a GameObject is placed on.

void Awake(): set all the start values.

void Update(): call GetInput() and Turn().

void FixedUpdate(): call Run() and Jump().

void GetInput(): save the user input for going forward, sideways, for turning, and for jumping.

void Run(): change the Rigidbody's x- and z-velocity depending on the input.

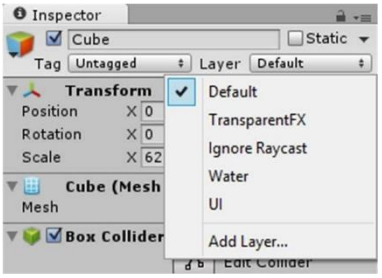
void Turn(): change the Transform's rotation depending on the input.

void Jump(): add force to the Rigidbody depending on the input and on whether the GameObject is currently grounded.

```
public class MoveSetings { }
public class InputSettings { }
```

```
[System.Serializable]
public class MoveSettings {…}
```

Note: You can also use Serializable in order to display private variables in the inspector, which would otherwise not be shown.



At first, set the LayerMask in the Inspector to “Everything” as all your GameObjects currently are in the Default Layer.

- Adjust the values of your different velocities later depending on your preferences.

In InputSettings we want to store the different axes of movement and input:

- There are up to four different axis that represent our input and the character’s movement:
- forward, sideways, turn and jump. We will save them in strings of the form: public string FORWARD_AXIS
- Set them to default values, which you can later change in the inspector. These values refer to Unity’s Input Manager.

```
public string FORWARD_AXIS = "Vertical";
public string SIDEWAYS_AXIS = "Horizontal";
public string TURN_AXIS = "Mouse X";
public string JUMP_AXIS = "Jump";
```

Create a reference for each class to access the values:

```
public MoveSettings moveSettings;
public InputSettings inputSettings;
```

Private Properties:

As we will move the GameObject using its Rigidbody component, we need a reference for it. To store the velocity we add a Vector3, and finally we need floats for the different kinds of user input.

```
private Rigidbody playerRigidbody;
private Vector3 velocity;
private Quaternion targetRotation;
private float forwardInput, sidewaysInput, turnInput, jumpInput;
```

Implement the functions:

Now, that all the preparation is done, we can implement the functions.

Let us start with Grounded().

We want to send a ray downwards in order to check, whether our character is standing on the floor using Unity’s Physics.Raycast function: public static bool Raycast(Vector3 origin, Vector3 direction, float maxDistance, int layerMask).

- The origin of the ray is the position of the GameObjects this script is attached to, the direction is downwards, the maximal distance is stored in our variable distanceToGround, and the LayerMask is determined by the variable ground (Unity takes here the integer value automatically).

```
Physics.Raycast(transform.position, Vector3.down,
        moveSettings.distanceToGround, moveSettings.ground);
```

- The origin of the ray is transform.position and it is sent downwards. Only if the distance between the origin and the hit object (which must be placed on our ground Layer) is smaller than distanceToGround, the function returns true – otherwise false. As our function Grounded() needs to return a boolean, you can simply add return in front of Physics.Raycast(…).

In Awake() we initialize our private variables.

- In the beginning, there is no user input, and the velocity is zero in all directions. There is no change in rotation; therefore, we will set targetRotation to the current rotation.

```
velocity = Vector3.zero;
forwardInput = sidewaysInput = turnInput = jumpInput = 0;
targetRotation = transform.rotation;
```

- To assign the GameObjects Rigidbody we need to make sure at first, that there is one attached.
  - Unity offers the command [RequireComponent (typeof (Rigidbody))] which will automatically add a Rigidbody to the GameObject, if there is none attached. Place it before your class PlayerBehaviour. o Go back to the Awake function and assign the variable:

```
playerRigidbody = gameObject.GetComponent<Rigidbody>();
```

Let's have a look at the different Update functions. Update() is called every frame and thus its interval times vary. It is used for moving non-physics GameObjects, or for receiving input. FixedUpdate() is called every physics step, meaning its intervals are consistent. (You can learn more in Unity's *manual*.)

In Update() we need to call GetInput() and Turn().

In FixedUpdate() call Run() and Jump(). Later we will add code to assign our velocity vector to the *Rigidbody's velocity property*.

Now we will implement GetInput() where we read and save the values of the different InputAxes.

- At first we will check if all the axes are used.
- Then, we will save the value of each input axis in the corresponding float value.

```
void Update()
{
    GetInput ();
    Turn ();
}
```

```
void FixedUpdate() {
    Run ();
    Jump ();
}
```

```
if (inputSettings.FORWARD_AXIS.Length != 0)
    forwardInput = Input.GetAxis(inputSettings.FORWARD_AXIS);
if (inputSettings.SIDEWAYS_AXIS.Length != 0)
    sidewaysInput = Input.GetAxis(inputSettings.SIDEWAYS_AXIS);
if (inputSettings.TURN_AXIS.Length != 0)
    turnInput = Input.GetAxis(inputSettings.TURN_AXIS);
if (inputSettings.JUMP_AXIS.Length != 0)
    jumpInput = Input.GetAxisRaw(inputSettings.JUMP_AXIS);
```

- The input values will be used in the Run, Jump and Turn functions.

In Run() we will change the x and z velocity depending on the values of forwardInput and sidewaysInput.

- To velocity.z and velocity.x we will assign a value depending on the variable runVelocity and the corresponding input value.

```
velocity.z = forwardInput * moveSettings.runVelocity; velocity.x  =
    sidewaysInput * moveSettings.runVelocity;
```

- Now apply the calculated values:

  playerRigidbody.velocity = transform.TransformDirection (velocity);
- Transform.TransformDirection(vector3 direction) takes the local direction (here our velocity vector) and finds the vector in world space. The velocity vector then is no longer in local space (relative to the GameObject itself), but in world space (relative to the game world).

Implement Turn().

- We will check if we have any turnInput, so we do not have to calculate a new value every frame.
- For the rotation we will use the function Quaternion.AngleAxis, which creates a rotation of a certain amount of degrees (first parameter) around a specified axis (second parameter).
- We will multiply the existing rotation with the new one in order to rotate by the rotation created by Quaternion.AngleAxis.
- We will set transform.rotation to targetRotation to apply the changes.

```
if (Mathf.Abs(turnInput) > 0)
{
    targetRotation *= Quaternion.AngleAxis(moveSettings.rotateVelocity *
      turnInput * Time.deltaTime, Vector3.up);
}
transform.rotation = targetRotation;
```

The last function we need to implement is Jump(). We need to check if there is any jump input, and whether the character is currently grounded, as we do not allow any double jumps. If this

is the case, we change the Rigidbody's y velocity and leave the x and z velocity as they are:

```
if (jumpInput != 0 && Grounded())
{
    playerRigidbody.velocity = new Vector3(playerRigidbody.velocity.x,
      moveSettings.jumpVelocity, playerRigidbody.velocity.z);
}
```

Now go back to Run(). Set the y-component of your Vector3 "velocity" to the Rigidbody's y velocity before
`playerRigidbody.velocity t = transform.TransformDirection(velocity).`
You need to do this because you changed the Rigidbody's velocity directly when jumping, and now want keep the y value when running.

```
velocity.y = playerRigidbody.velocity.y;
```

That is it. Attach the script to the Player GameObject (if you have not already done so). Expand

"Input Settings" in the Inspector and make sure it reads "Mouse X" and not "MouseX". Expand "Move Settings" and set the LayerMask ground to "Default". Set the Layer of the Player to "IgnoreRaycast".

Go try it out!
If you do not like mouse control, you can easily change this in the Inspector. For example, you can change TURN_AXIS from Mouse X to Horizontal and leave SIDEWAYS_AXIS empty.

Further Reading:

- This is only one way of processing input; you can also use different Rigidbody functions to move the character, Unity's *CharacterController* component or manipulate the Transform component.
- If you are interested in different jump mechanics or try another, maybe more advanced one, have a look at this *this* blog post.

Camera Follow Script

Your character can move now, but the camera is not following yet. We will handle this by adding a new script to the main camera.

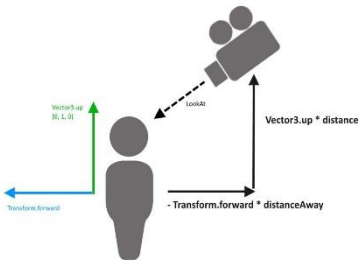
- Create a new C# script "CameraFollow" and open it.
- We will need variables to store the camera's target (a Transform) which it should follow, and the distance both in the

away and up direction (floats). To store this position we will declare a Vector3 variable.

- To smooth the camera's movement we need another float variable.

```
[SerializeField]
private float distanceAway;

[SerializeField]
private float distanceUp;

[SerializeField]
private float smooth;

[SerializeField]
private Transform followedObject;

private Vector3 toPosition;
```

- All these variables can be private as we do not want to modify them from other scripts. To be able to nevertheless see them in the Inspector, add [SerializeField] before each of them, if you want to see them in the Inspector.
- We will only need a LateUpdate function. LateUpdate() is called after Update() has finished, so this ensures that the character has moved completely before the camera tracks its position.
- In LateUpdate() we will determine the targetPosition of our camera. It depends on distanceUp and distanceAway.
- Set the camera's new position to targetPosition. But do not do this at once, use *Vector3.Lerp* instead, which allows linearly interpolates between two vectors.
- Now, you only need to make sure, the camera looks at the target.




```
toPosition = followedObject.position + Vector3.up * distanceUp –
    followedObject.forward * distanceAway;
transform.position = Vector3.Lerp (transform.position, toPosition,
    Time.deltaTime * smooth);
transform.LookAt(followedObject);
```

Attach the script to the Main Camera. Assign the Player to the Transform followedObject, and add these example values for the script variables:

- Distance Away: 5
- Distance Up: 2
- Smooth: 100

Of course, there are more advanced ways of having the camera follow the player, and if your world design requires it, you can (and probably should) also handle camera collision to make sure your camera does not simply go through objects. Keep this in mind for when you make your own games!

Professur für Erweiterte Realität
Fakultät für Informatik
Technische Universität München

Homework (do after class)

Make your level more enjoyable

Homework 1: Add different Platforms
To expand your level and make it more appealing, add different platforms to your game (at least two different types). For example, a platform that

- moves between more arbitrary points
- rotates around its center
- rotates around a specific point
- can become invisible after a certain time
- falls down after a certain time
- …

Interaction between Player and Platforms

The final goal of the tutorial is to build a complete 3D Platformer Level. Make sure you have a character which you can control and further adapt the platforms that you have created before!

Homework 2: Character & Camera Variables
Make sure you have a character which you can now control and a camera which follows the character accordingly. Play around with variables to find suitable speeds for the movement of the player and camera.

Homework 3: Make Platforms Interactable
- You might have noticed that when a player stands on top of a moving platform, the player would stay at the same place and eventually falls.
- Think of a solution, so whenever the player stands on a moving platform, the player is moved with the platform as well.
- Give the player the ability to destroy platforms or place platforms
- Be creative

Technische Universität München
Fakultät für Informatik
Professur für Erweiterte Realität

Boltzmannstr. 3
85748 Garching bei München

https://far.in.tum.de