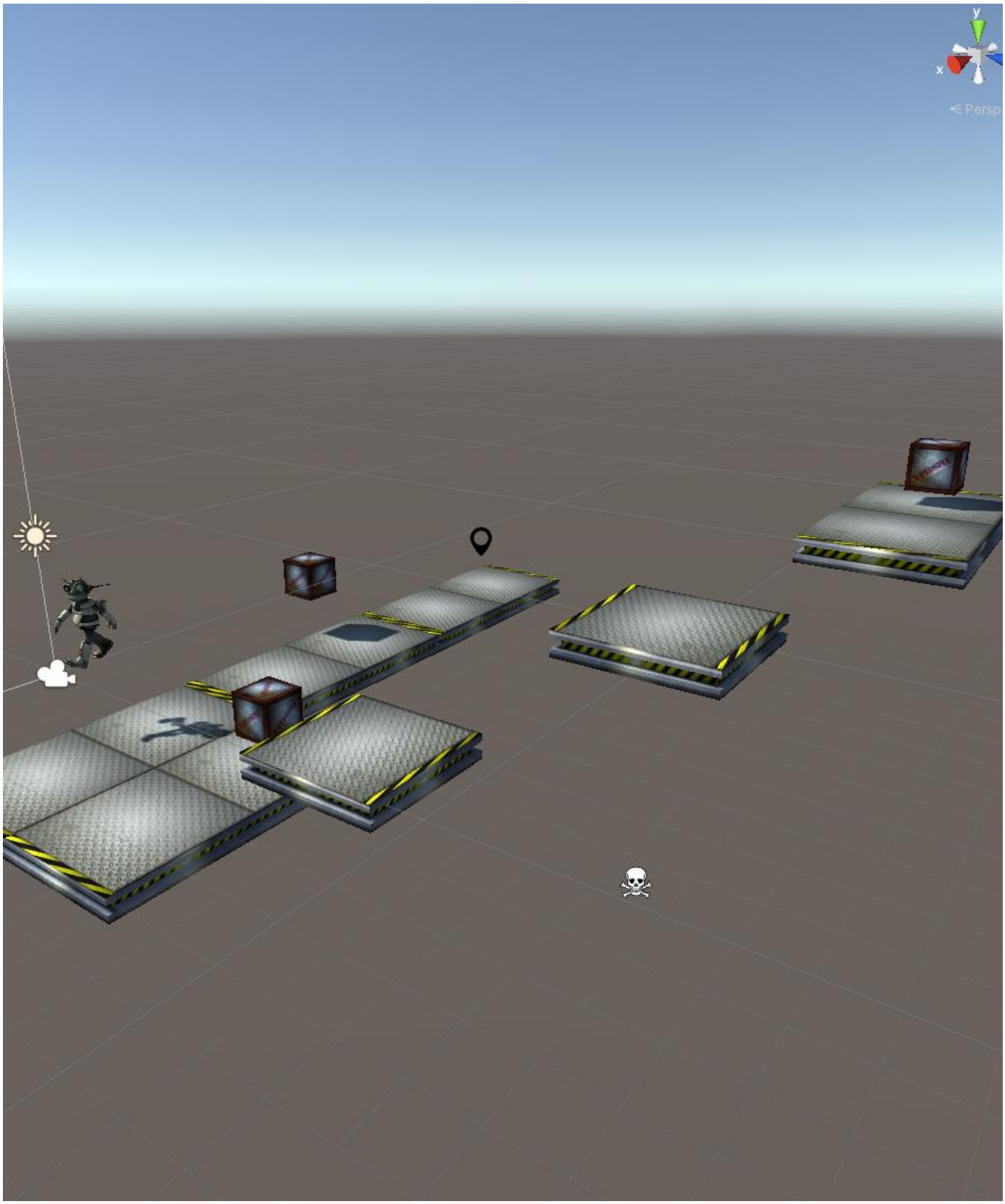
Professur für Erweiterte Realität
Fakultät für Informatik
Technische Universität München

Tutorial & Exercises – Tutorial 7

Platformer



Invisible GameObjects and Gizmos: Creating a Death Zone and a Spawn Point

Deathzone

You have set up platforms and your character can walk on them. But what if the player falls down? To handle this we will add another functionality to the basic setup of your scene: a death zone.

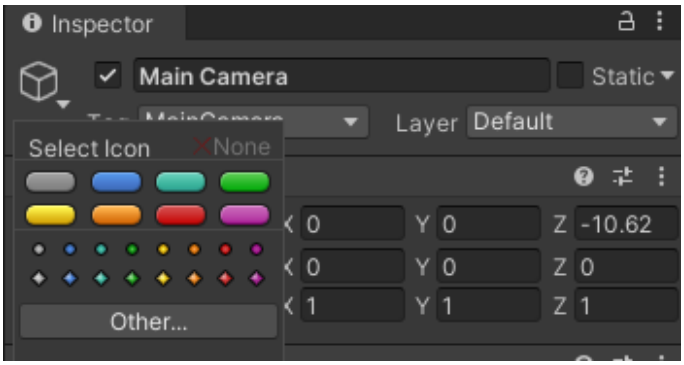
The death zone must consist of a trigger and must be identifiable so that the player GameObject can check if it enters the death zone:

- Create an Empty GameObject: GameObject Create Empty
- Add a Box Collider component and mark it as trigger.
- Adjust the size of your death zone and place it below you platforms.
- Create a new tag “DeathZone” and apply it to your GameObject.

The death zone is not visible as it has no mesh geometry nor a mesh renderer. You can use a gizmo to know where the death zone is located.

Unity offers two possibilities to add gizmos.

Using the Icon Selector in the Inspector, you can easily select custom items. They will be displayed at the GameObjects position.



Gizmos

Gizmos are shapes, icons and other visual aids that appear in the Scene View. They assist in visual debugging or layout.

To add more elaborate gizmos use one of Unity's gizmos functions.

- `void OnDrawGizmos()`: The gizmos in here will always be drawn.
- `void OnDrawGizmosSelected()`: The gizmos in here will only be drawn when the object is selected.

We will use OnDrawGizmos() to permanently draw an icon (a texture) at the death zone's position.

- Create a new script "DeathZone", attach it to your DeathZone GameObject, and open it.
- Delete the Start and Update functions as we will not need them. Instead add the `OnDrawGizmos` function and call `public static void DrawIcon(Vector3 center, string name, bool allowScaling = true)` on Gizmos.
- The centre parameter denotes the location of the icon in world space while the image filename for the icon is specified with the name parameter.
- The texture with the name "deathzone" must be in a folder called "Gizmos". Create it in your Assets folder and move the image file there.

```
public void OnDrawGizmos()
{
    Gizmos.DrawIcon( gameOb-
    ject.transform.position,
    "deathzone");
}
```

Further Reading: Of course, you can draw a lot of other visual aids. Have a look at the documentation.

Exercises (Prepare before class)

Building a small level
Create an appropriate death zone with gizmo icon in your scene

Professur für Erweiterte Realität
Fakultät für Informatik
Technische Universität München

Tutorial (do within class)

Spawn Point

When you start your game, you want the player character be at exactly the same position each time, regardless of where you place the GameObject in the scene. This position will be the spawn point:

- Position Create a new empty GameObject: GameObject
- Create Empty. Rename it to “Character Spawn Point”.
- Create a new C# script titles “SpawnPoint” and let it display a Gizmo Icon (just as you did with the death zone). Apply it to the GameObject you just created.
- Position the spawn point where you want you player character to appear.

```
public void OnDrawGizmos()
{
    Giz-
    mos.DrawIcon(gameObject.trans
    form.position, "spawnpoint");
}
```

Now you must add a spawn functionality to your player script. You can simply extend PlayerBehaviour.cs:

- Add a field in which you store the spawn point:
- Add a function public void Spawn() in which you set the player GameObject’s to your spawn points position.

```
public Transform spawnPoint;
```

- Add a Start function and call Spawn() in it. (Of course, you could write the code directly in the Start function, but you want to reuse it later. By transferring the code to a separate function you avoid repetitive code.)
- Do not forget to assign the Character Spawn Point GameObject to the corresponding variable slot in the script.

```
public void Spawn()
{
    transform.position = spawn-
    Point.position;
}
```

Respawn on Death

To handle the collision between the player and the DeathZone create a new function in PlayerBehaviour.cs:

- In OnTriggerEnter we will check if the Collider we entered is the death zone.
- Then we will call the new Spawn function to reset the player.
- Make sure that the DeathZone-GameObject has the tag “DeathZone”.

```
void OnTriggerEnter(Collider other)
{
      if (other.gameObject.tag ==
      "DeathZone")
      {
        Spawn();
      }
}
```

Editor Scripting: Menu Items

We will have a look at editor scripting to create a simple and understandable game editor that is suitable for your game. Unity offers a lot of different and special possibilities of editor scripting, and we will introduce you to the basics, so you can (more) easily implement the editor items you need in later games.

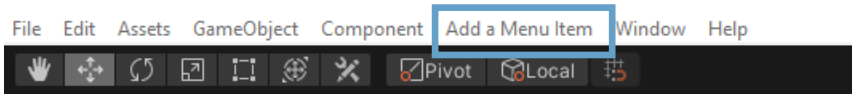
You can change the Inspector, add windows or adjust the menu bar, just to name a few examples. As an example we will add Menu Items.

The Basics of Editor Scripting
Regardless of the type of editor scripting you do, you will need a folder “Editor” that contains all your editor scripts. It is a special name that indicates the usage of the folder (similar to the “Gizmos” folder). This also makes sure, Unity does not include this folder in the actual game build.

In this folder, locate all your editor scripts. Each editor script must have the following statement on top of the code: `using UnityEditor;` It should look like this:

```
using UnityEngine;
using UnityEditor;
using UnityEditor.SceneManagement
using System.Collections;
```

Menu Items
Now we can start with the editor functionality we want to implement: Menu Items.



We want a button that will open another scene and save the current one. For each scene we need to add a new function, but we only have to do this once and then benefit for the rest of the development process.

Let us assume we have two scenes: a start scene, and a level scene (which you already have). Add the missing scene to your game and name it “StartScreen”. Don’t forget to add all your scenes to the Build Settings. Make sure, they both are in a folder “Scenes” in “Assets”.

Create a new C# script “OpenScene” in your Editor folder which will contain the Menu Item that opens another scene.

- Add – as explained above – `using UnityEditor`.
- Delete the Start and Update functions as we will not need them.

```
[MenuItem(“OpenScene/Start Screen”)]
```

- Add a new line of code which adds a new Menu Item (The string within the brackets defines the hierarchy: “Start Screen” is a subitem of “OpenScene”.):
- The function that follows this line of code will be executed whenever you click on “Start Screen”. We will add the function which will open a window that asks the user whether to save the current scene and then will open the scene “Start Screen”. The function must be `static`.
- Copy and paste the code and adjust it for all your other scenes.

```
static void StartScreen()
{
    EditorSceneManag-
    er.SaveCurrentSceneIfUserWantsTo ();
    EditorSceneManager.OpenScene ("As-
    sets/Scenes/StartScreen.unity");
}
```

- If you want to add keyboard shortcuts to each subitem, you can simply add to the string above each function: “ %0” for cmd-0 (replace the “0” with the key for your shortcut, e.g. a number for each scene).

```
[MenuItem(“OpenScene/Start Screen
    %0”)]
```

Assuming you have three different scenes, the following picture displays what your Menu Item should look like. Each subitem will ask the user whether to save the current scene and will open the respective scene.

Further Reading: This is an example of how editor scripting in Unity works. If you are interested in different ways to customize your editor, have a look at Unity’s tutorial videos and the related tutorials.

Homework (do after class)

Collisions and Rigidbodies

The final goal of the tutorial is to build a complete 3D Platformer Level. Learn about collisions and rigidbodies so you can add interactable objects in your game.

Exercise 1: Adding Obstacles and Objects

Rigidbodies are the gateway to applying physics to your objects. You already know how to handle collisions and how to move a Rigidbody.

Now we will use it for an interactable obstacle: a crate. Its physics components will be:

- Box Collider
- Rigidbody

Add as many crates (Cubes) to your scene as you want. Play with the different settings. For example, make crates that the player can push, or ones that cannot be moved. Try changing the collider to a Sphere Collider and examine what happens. You can also restrict the Rigidbody's movement to certain axes. What differences occur when you (un-)check Use Gravity and Is Kinematic?

Further Reading: If you want to add a constant force to your Rigidbody, you can add the physics component Constant Force.

Usage
Using Rigidbodies and/or colliders you can create many different elements for your game. The way your player character interacts with them among others depends on whether you use a collider or a trigger. Examples for interactive objects are

- moveable crates (the player can push and pull them, e.g. in order to overcome obstacles)
- barriers and boundaries
- collectibles (gems, keys, medi kits…)
- checkpoints

- trigger zones for events like change of an NPC's animation state, teleports, opening doors, changing the direction of a moving platform…

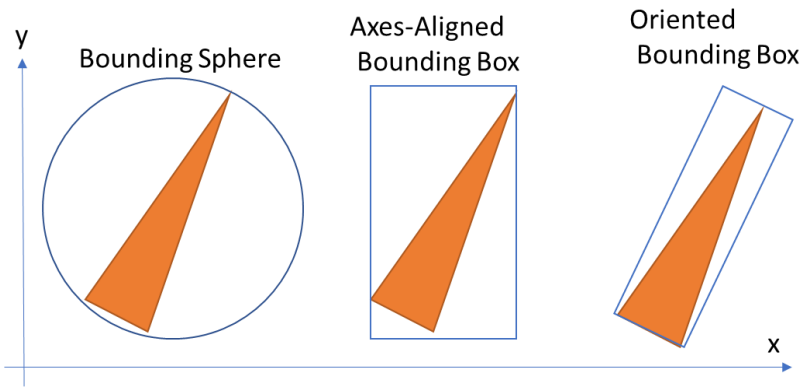
Think of other possible applications and how to use them in your game (and add the ones that support your game's mechanics or mood).

Exercise 2: Reading Exercise

Collisions

The goal is to detect physical contact / collision in real-time. We already know how to handle this by code: Unity provides the functions `OnTrigger…` and `OnCollision…` (Enter, Stay and Exit). We need to make sure, that all the objects that we want to be able to detect collision have a collider (and a Rigidbody) attached.

But how does the choice of a bounding volume influence the collision detection?

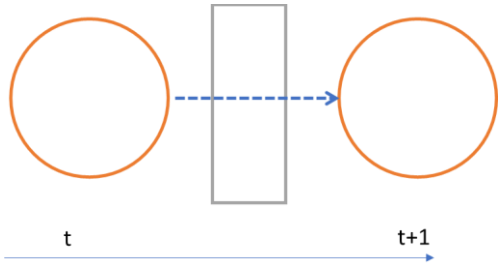


Usually your colliders are oriented.

Choose the best bounding volume for your `GameObject`. If Unity's basic collider volumes are not detailed enough, you can also use a mesh collider. But keep in mind that only convex mesh colliders can collide with other convex colliders and non-convex meshes.

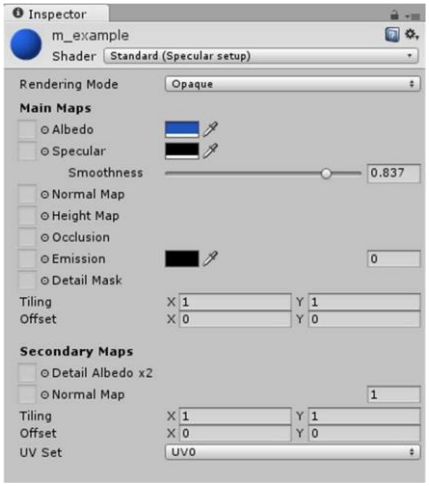
The default collision detection mode is discrete. Collisions for this collider will only be checked at the content's `Time.fixedDeltaTime` (like MonoBehaviours's `FixedUpdate`). If your object is moving

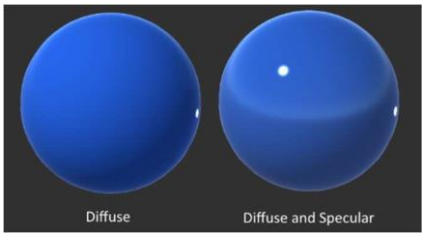
very fast, change the collision mode to continuous to make sure collisions are detected, even if they occur between two FixedUpdate steps:



Check Unity's [script reference] for more details.

Appearance



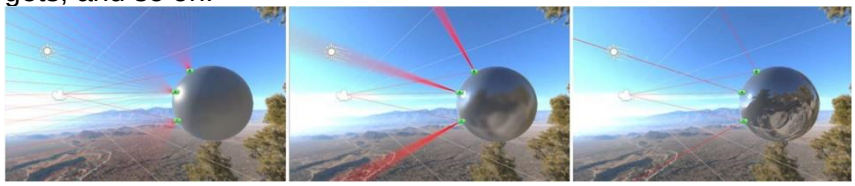


Shaders, Materials and Textures
You have read about tiling and you already know how to use materials. Now, we will delve deeper into Shaders, Materials and Textures.

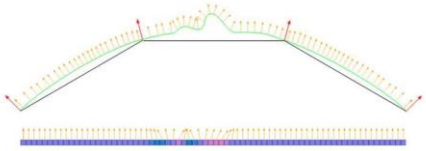
Materials:
Materials are definitions of how a surface should be rendered, including references to textures used, tiling information, color tints and more. The properties that a Material's inspector displays are determined by the Shader that the Material uses.

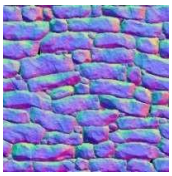
Shaders:
A shader is a specialized kind of graphical program that determines how texture and lighting information are combined to generate the pixels of the rendered object onscreen.

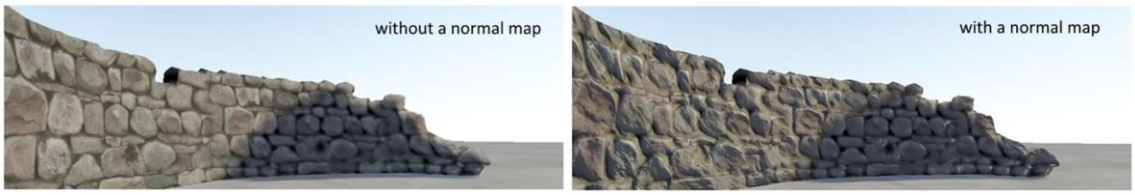
Light reflection:
Unity has two [Standard Shaders]: a specular approach and a metallic one.
The less smoothness is applied, the more diffuse the reflection will be; and the smoother, the sharper the reflection. The specular setup lets you set a different color for the specular reflection. The more specular a material is, the less diffuse it should be; the smoother a surface is, the stronger and smaller the highlight gets, and so on.



Normal Mapping:
Normal maps are a special kind of texture that allow you to add surface details. No geometric displacement is needed as the normal map will simulate the variety of normals. The x-, y-, z-coordinates of calculated normals will be stored in the RGB-components of the texture.







http://docs.unity3d.com/Manual/StandardShaderMaterialParameterNormalMap.html

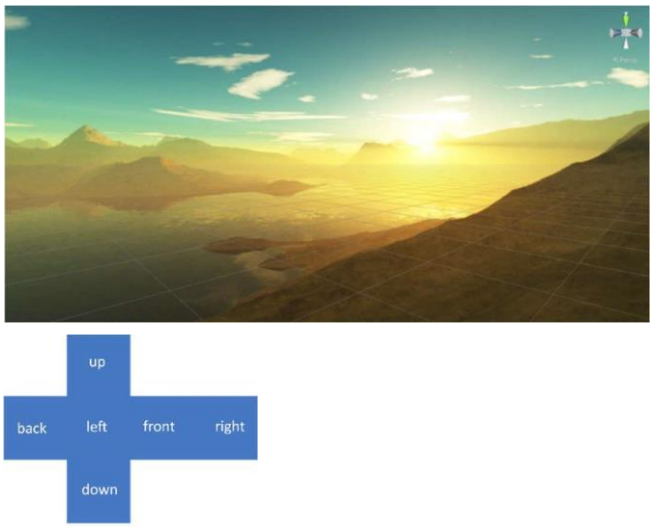
What is the advantage of using a normal map (instead of adding the detail in the geometry)?

Lighting
To create the mood of your scene, we will add lighting. Light GameObjects are necessary to calculate the shading of a 3D object as they provide intensity, direction and color of a light. Unity provides four different light types:

- Point Lights: A Point Light is located at a point in space and sends out light in all directions equally.
- Spot Lights: A Spot Light has a specific location and range. It is constrained by an angle, resulting in a cone-shaped region of illumination.
- Directional Lights: A Directional Light does not have any identifiable source position and can so be placed anywhere in the scene. All objects are illuminated as if the light is always from the same direction.
- Area Lights: An Area Light is defined by a rectangle in space. Light is emitted in all directions, but only from one side of the rectangle.

Skyboxes
Skyboxes are a wrapper around your entire scene that shows what the world looks like beyond your geometry.
Skyboxes are implemented through Skybox shaders:

- You can create a new material and select one of the three Skybox shaders.
- The example on the left is the 6 sliced one. As you can see, each texture represents one of the main axes. Combined they make up the skybox.



- Today there are also different types of 360° images using different reprojections (read more).
- Skyboxes are part of the environment lighting: Window -> Lighting.

Technische Universität München
Fakultät für Informatik
Professur für Erweiterte Realität

Boltzmannstr. 3
85748 Garching bei München

https://far.in.tum.de