UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

PROJECT

New! 2022

3D Augmented Reality

A.Y. 2022/2023

NavMesh, Raycasting & Simulation

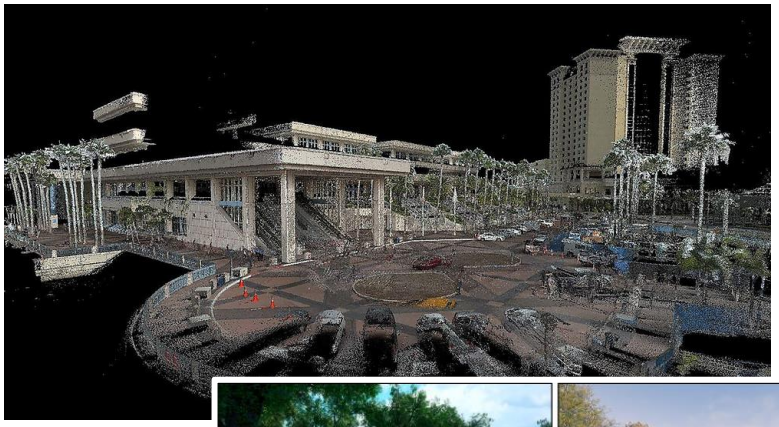
LAB experience 5

Elena Camuffo

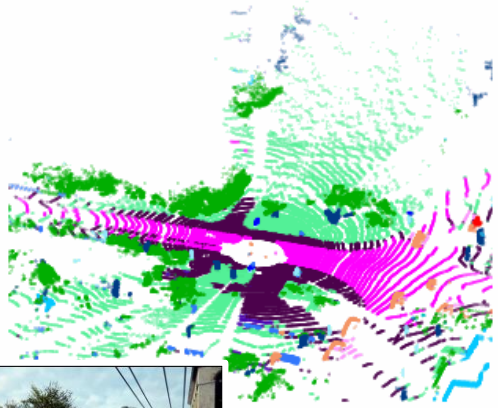
elena.camuffo@phd.unipd.it

Point Cloud types

- Nowadays, there are three main acquisition methods for point clouds that provide completely different point cloud data: **static, sequential and synthetic**.



Static

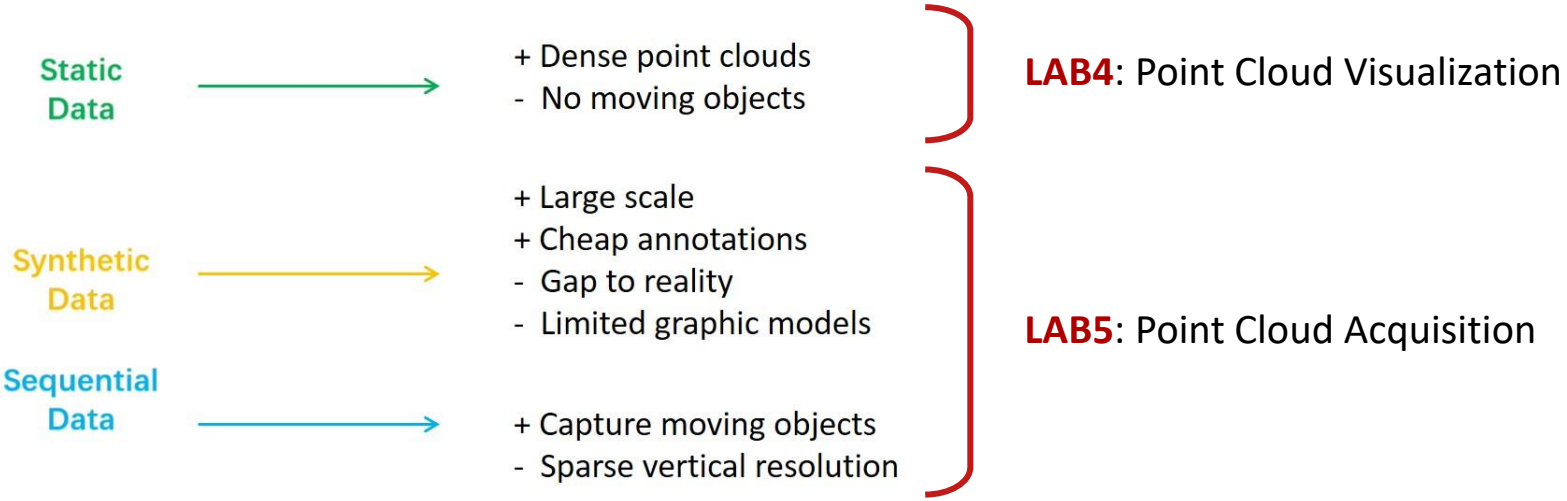


Sequential

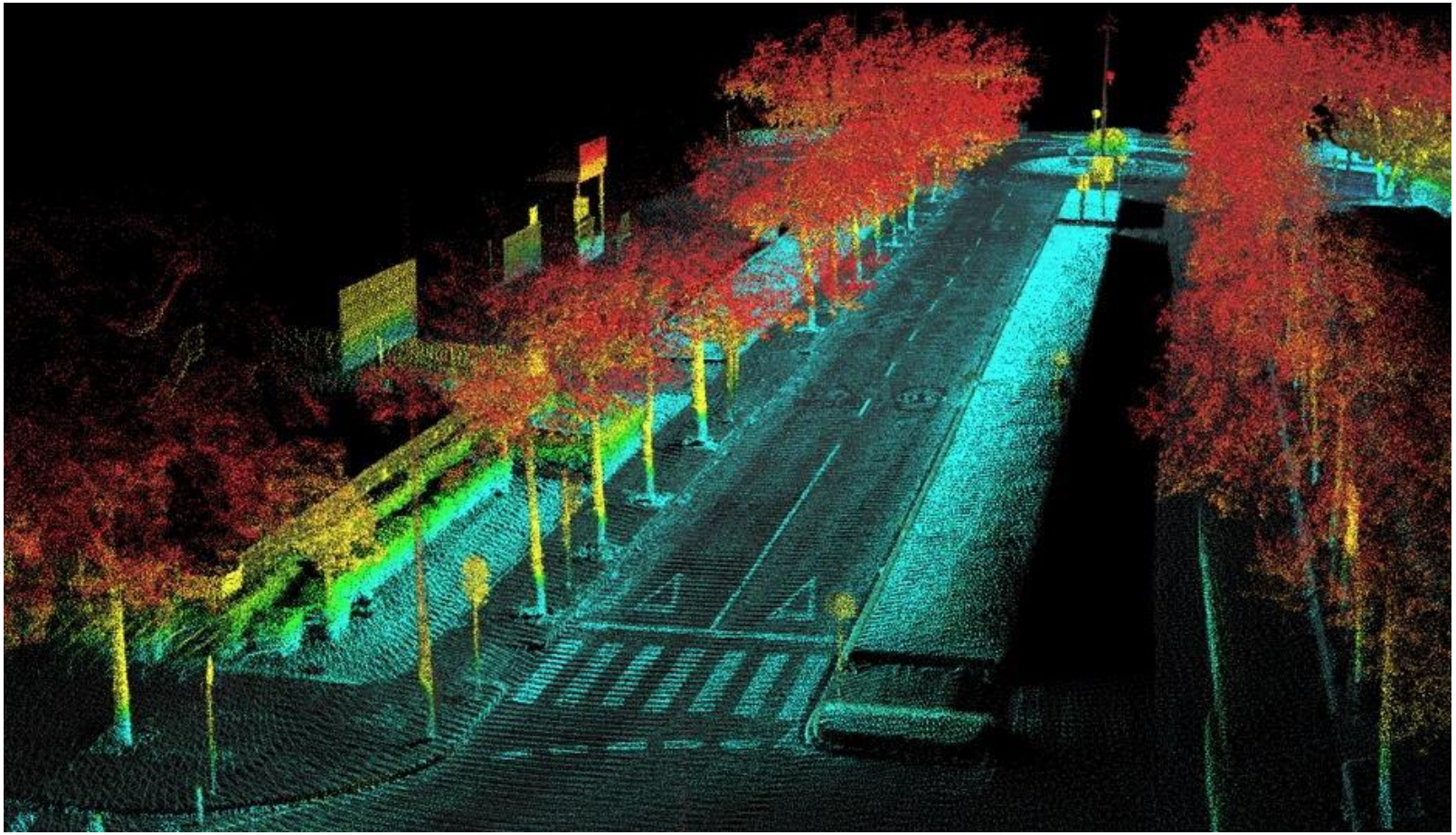
Synthetic

Point Cloud types

- These typologies present very different characteristics and are used for different applications.
- **Static Data** are huge point clouds usually representing complete 3D scenes.
- **Sequential Data** are acquired sequentially with sensors like LiDAR scanners and mostly used in autonomous driving.
- **Synthetic Data** can be either static or sequential, but are generated within a simulation environment (*e.g.,* Unity3D).

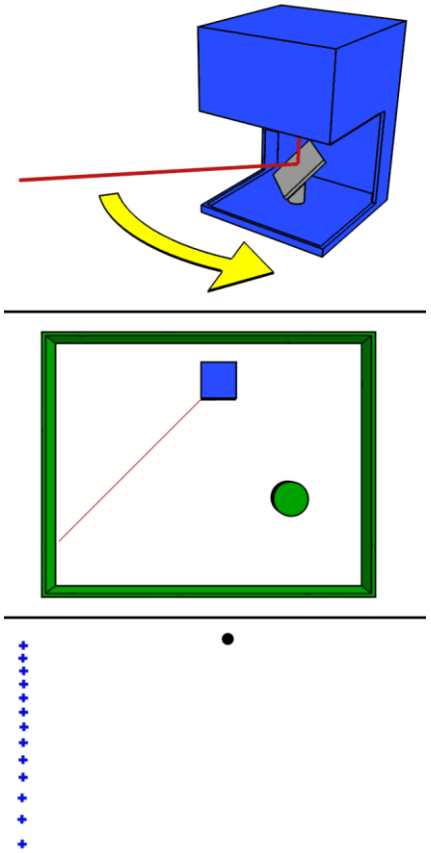


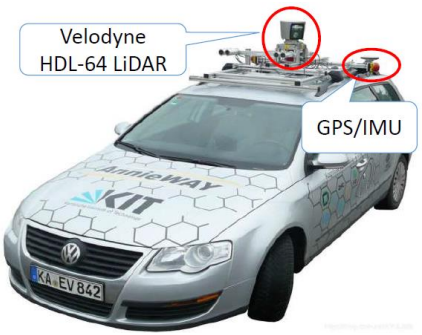
LiDAR Point Clouds



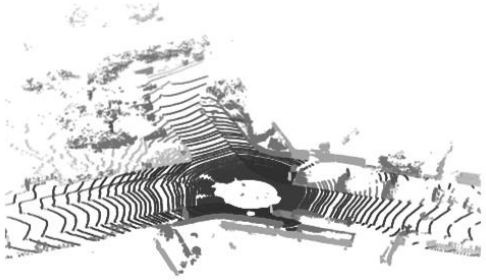
LiDAR Acquisition Sensor

- Light Detection and Ranging or **Laser Imaging Detection and Ranging (LiDAR)** are detection systems which work on the principle of radar but use light from a laser.
- These sensors produce a **sparse prediction** of the environment composed of point cloud data.





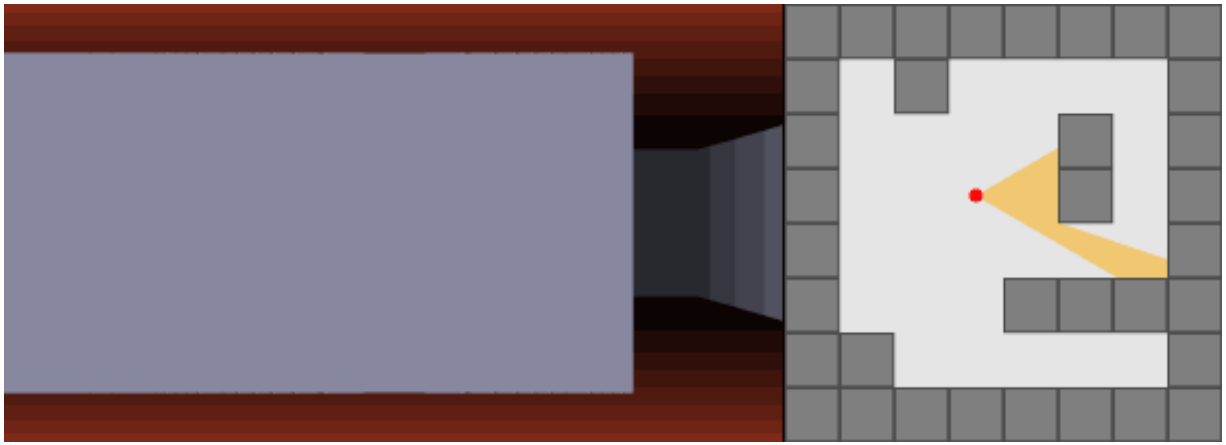
acquisition system

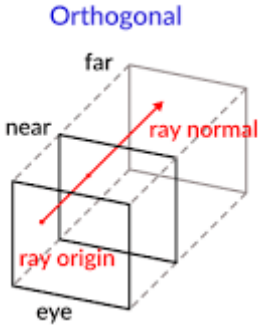


example point cloud acquired

Raycasting

- In 3D computer graphics, **ray tracing** is a technique for modeling light transport for use in a wide variety of rendering algorithms for *generating digital images*.
- **Raycasting** is the simplest form of ray tracing, mostly used for CGI.

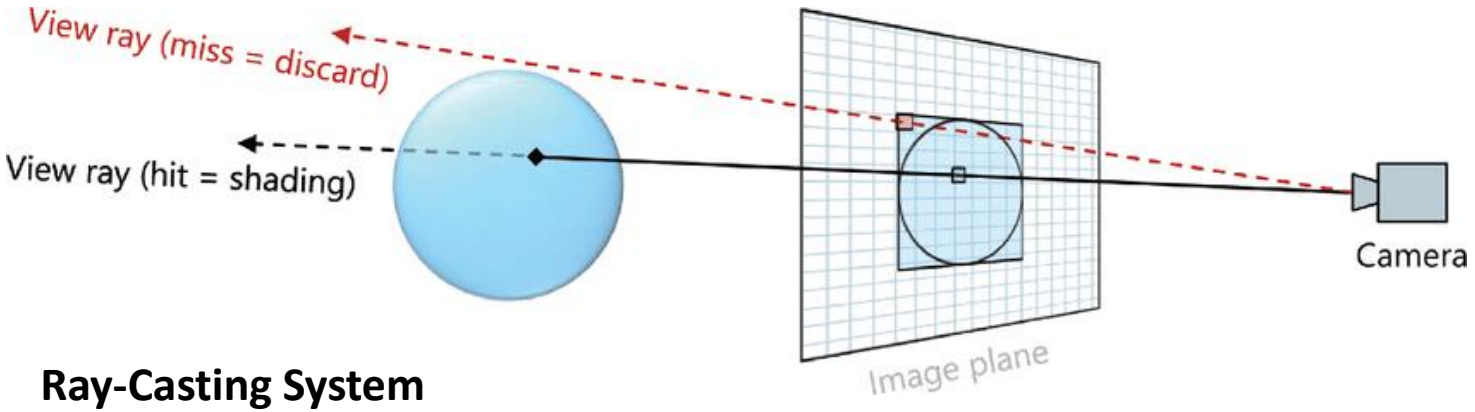




- A raycast sends an imaginary *laser beam* along the ray from its origin **until it hits a collider** in the scene. Information is then returned about the object and the point that was hit.

Raycasting

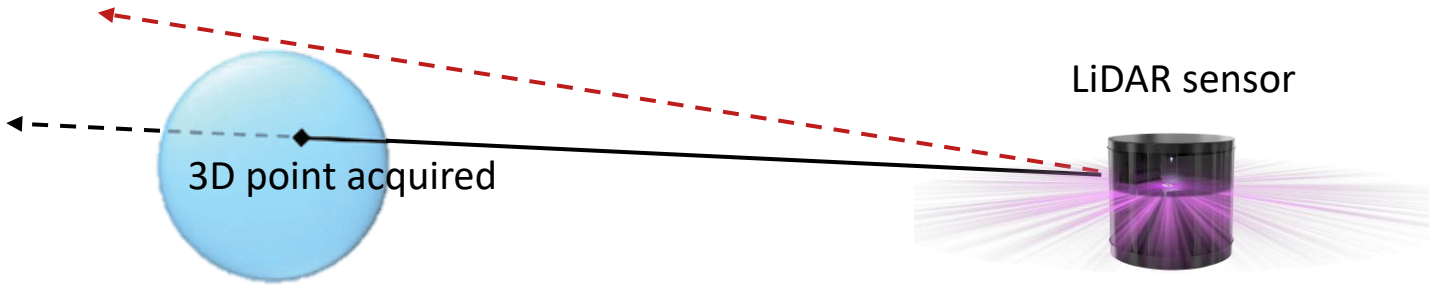
- Unlike other forms of ray tracing, where rays originate from a light source and bounce off objects to arrive at the observer, in ray casting, they're ***cast* directly from the viewpoint**.
- When cast rays intersect an object, the object's **color and brightness** at that point determines the value of one pixel in the final image.
- A direction vector represents the orientation of the observer extending forward. An *image plane* perpendicular to the vector, represents the shape of the final rendered image.



Ray-Casting System

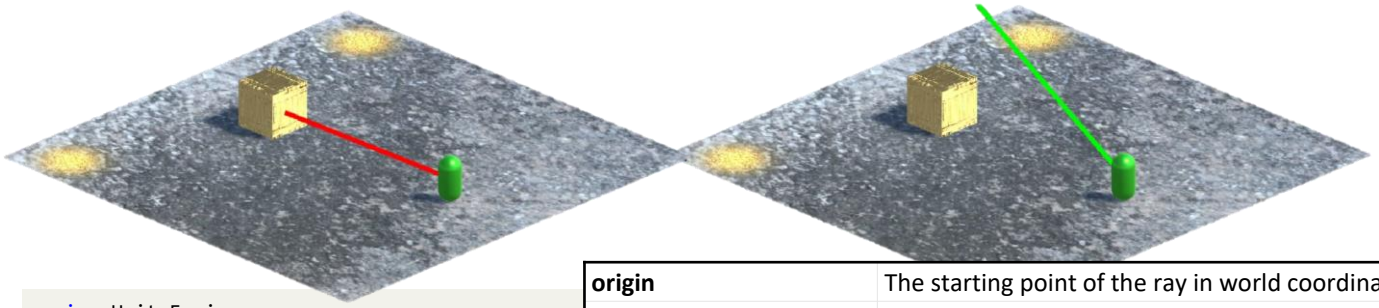
Simulating LiDAR Sensor through Raycasting

- Also in **LiDAR sensors**, a bunch of *rays departs from the viewpoint* to hit the scene external surfaces and detect objects.
- Past approaches to simulation have included methods such as: *light masked projectors,* which emulates LiDAR with Unity lighting, but there is no way to determine point counts on objects; *particle-based systems* can address this problem but is very costly in computation time and memory.
- The best solution is to simulate LiDAR sensors in Unity through Raycasting.



Ray-Casting for
Simulating LiDAR in Unity

Script Example



Debug.DrawLine() works only in **Scene View**. Use *Line Rendered* component to draw lines in **Game View**.

Parameters

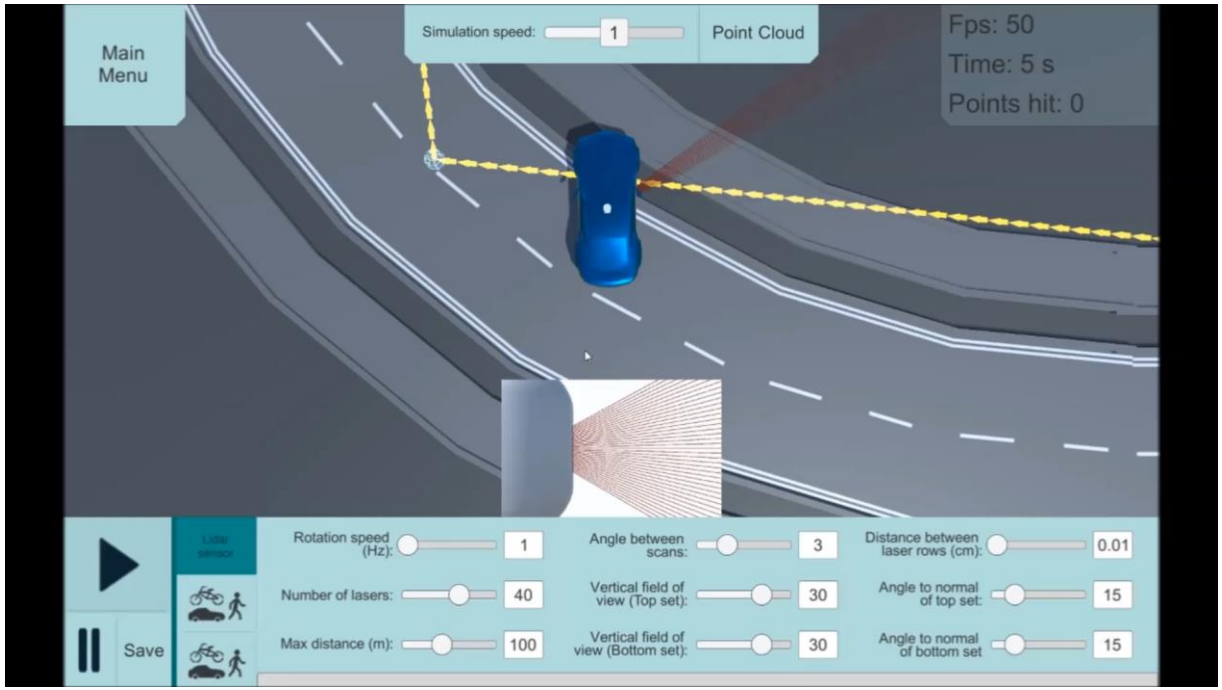
origin	The starting point of the ray in world coordinates.
direction	The direction of the ray.
hitInfo	Bool contains more information about where the closest collider was hit.
maxDistance	The max distance the ray should check for collisions.
layerMask	A *Layer mask* that is used to selectively ignore colliders when casting a ray.
queryTriggerInteraction	Specifies whether this query should hit Triggers.

```
1  using UnityEngine;
2
3  public class RaycastLiDAR : MonoBehaviour
4  {
5     // Layer mask
6     [SerializeField]
7     LayerMask layerMask;
8
9     void FixedUpdate()
10     {
11         RaycastHit hit;
12
13         // Does the ray intersect any objects excluding the player layer
14         if (Physics.Raycast(transform.position, transform.TransformDirection(Vector3.forward), out hit, Mathf.Infinity, layerMask))
15         {
16             Debug.DrawRay(transform.position, transform.TransformDirection(Vector3.forward) * hit.distance, Color.red);
17             Debug.Log("Did Hit");
18         }
19         else
20         {
21             Debug.DrawRay(transform.position, transform.TransformDirection(Vector3.forward) * 100, Color.black);
22             Debug.Log("Did not Hit");
23         }
24     }
25 }
```

LiDAR Simulator in Unity Example

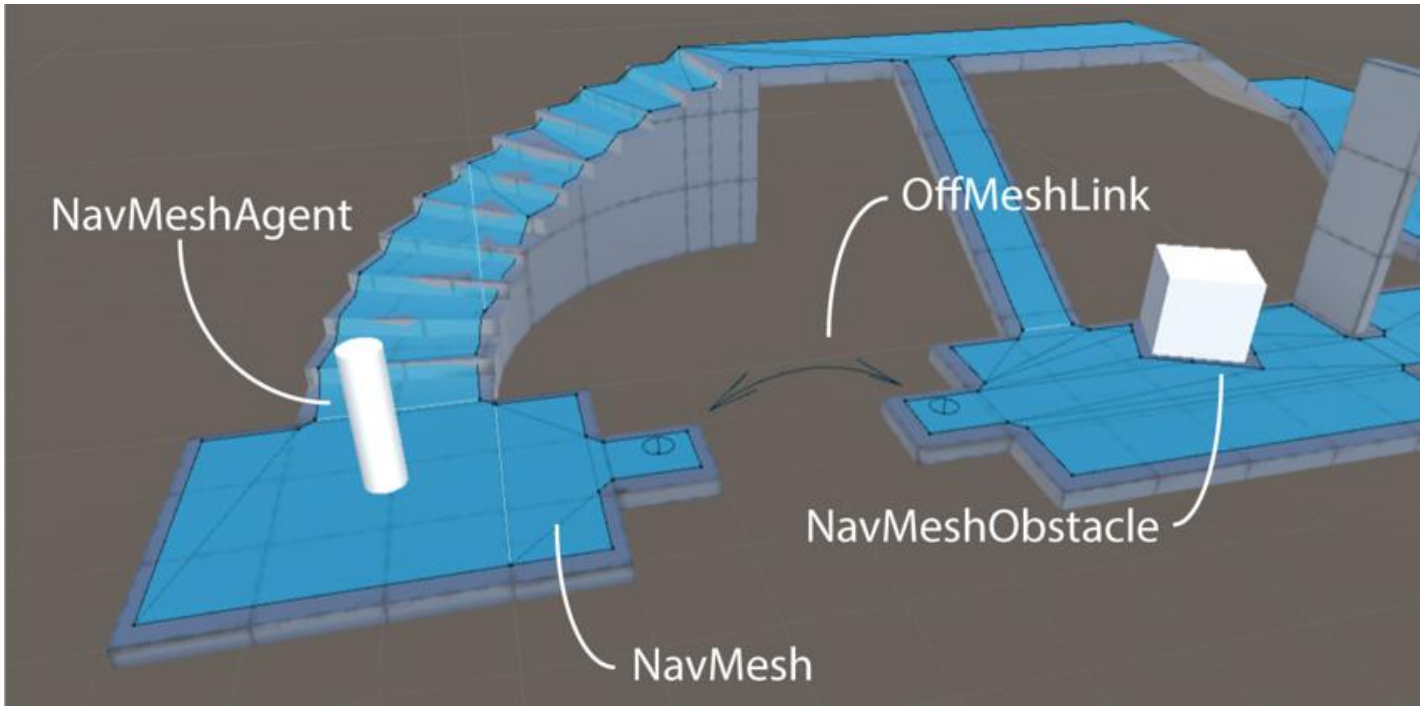
- This example is a complete environment for **simulating an autonomous driving scenario**.
- You can download the whole project cloning the following GitHub repository:

  https://github.com/ptibom/Lidar-Simulator



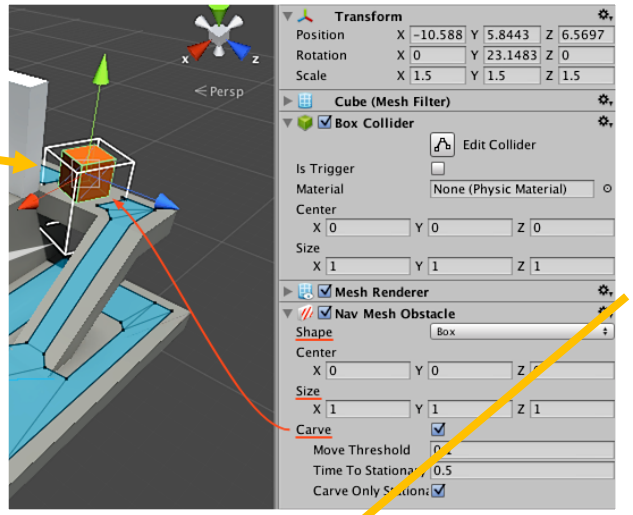
NavMesh

- The **Navigation System** allows you to create characters which can navigate the game world.
- **NavMesh** is a data structure which describes the walkable surfaces of the game world and allows to find path from one walkable location to another in the game world. The data structure is built, or baked, automatically from your level geometry.

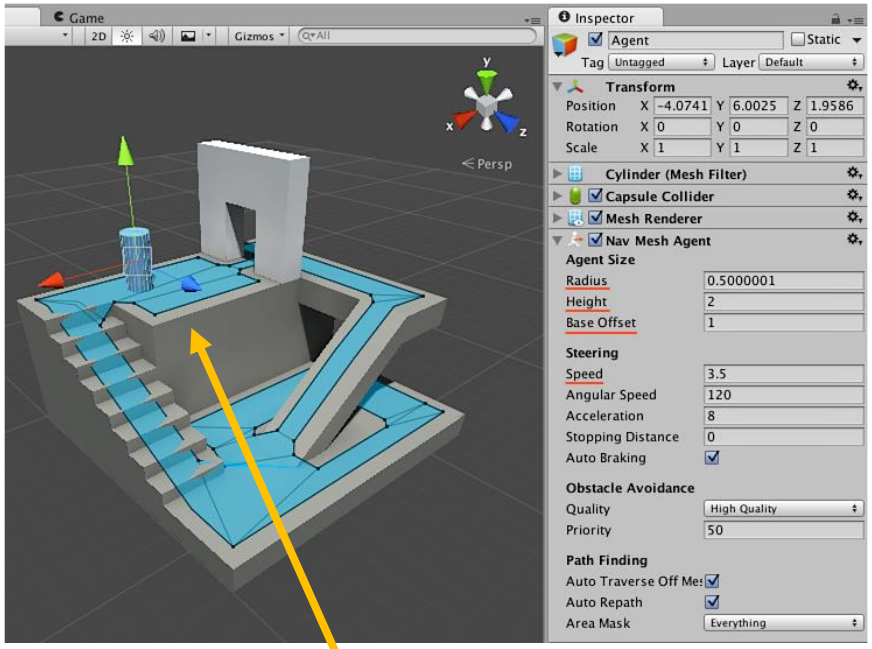


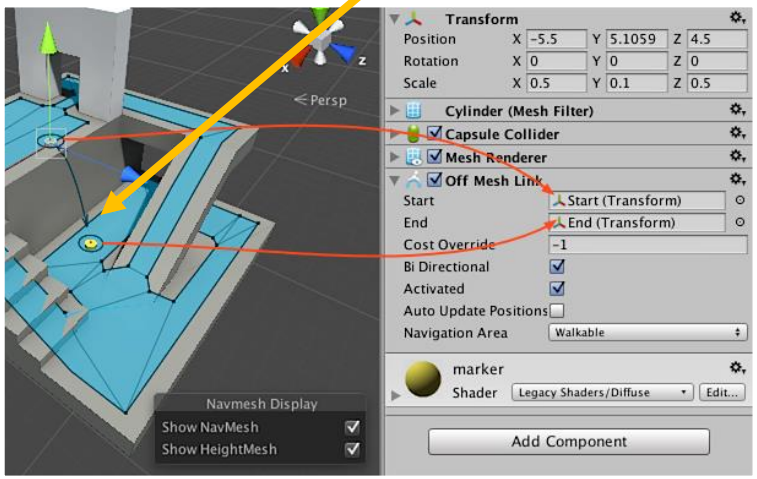
NavMesh Components

NavMesh Obstacle allows you to describe moving obstacles the agents should avoid while navigating the world.



Off-Mesh Link incorporate navigation shortcuts which cannot be represented using a walkable surface.

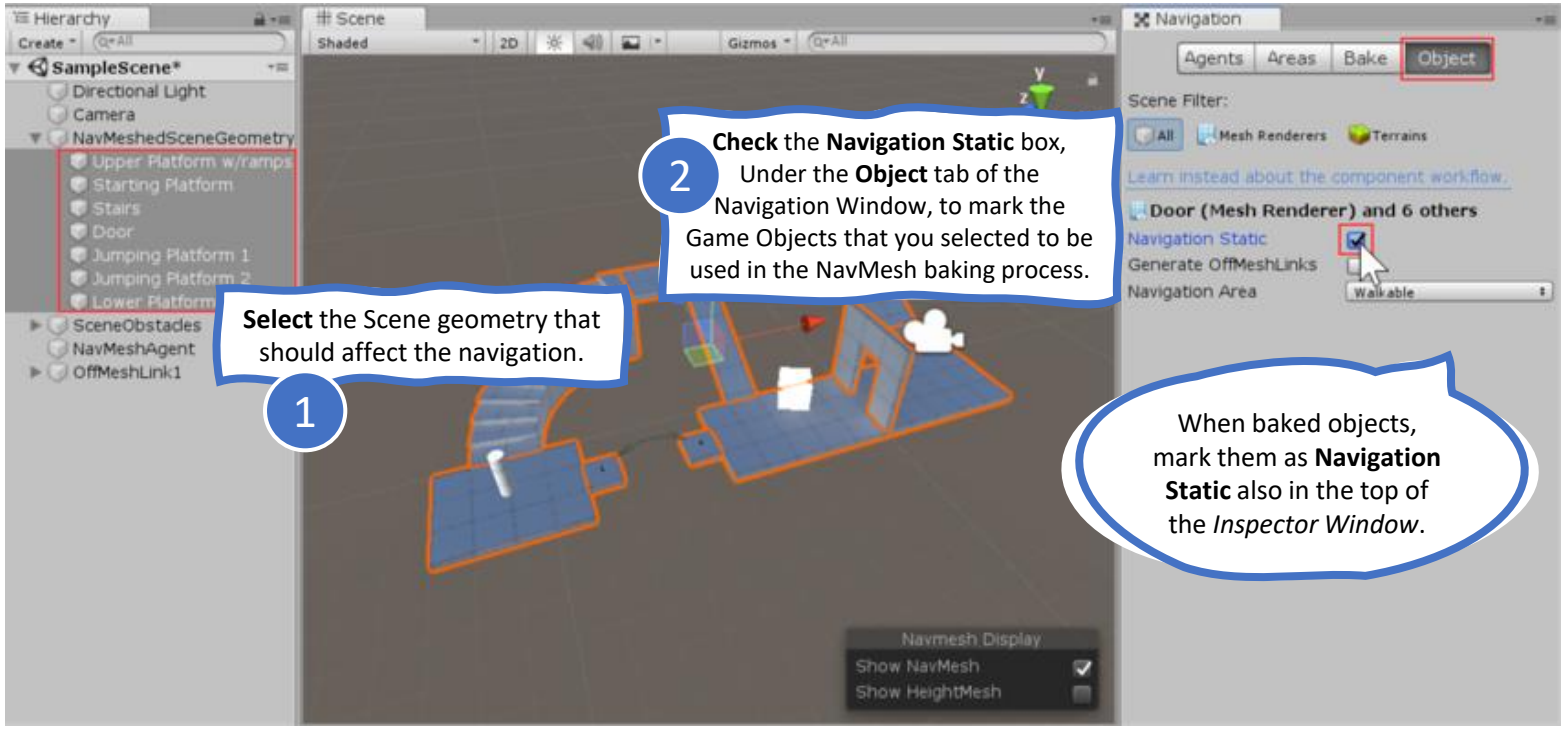




NavMesh Agents reason about the game world using the NavMesh and they know how to avoid each other as well as moving obstacles.

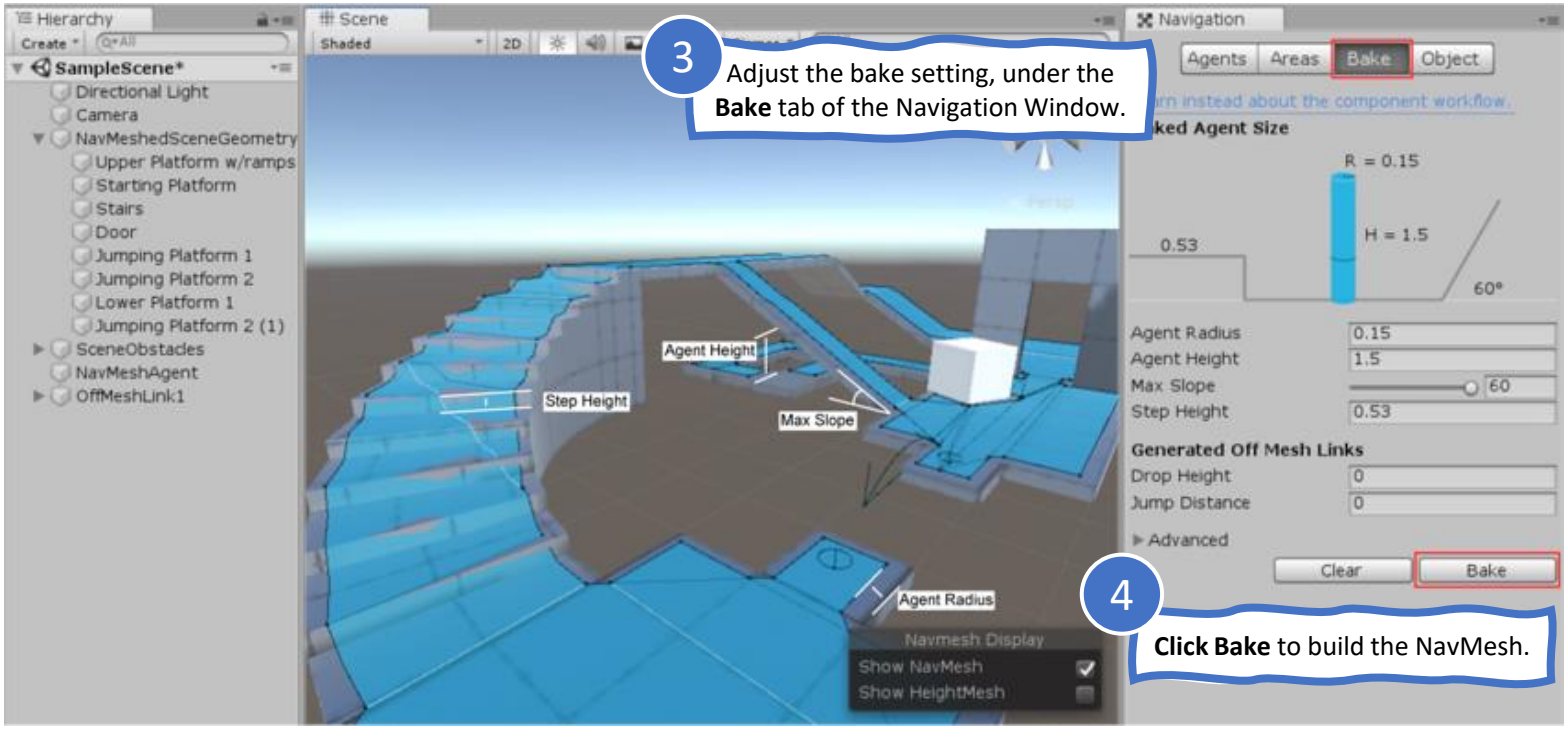
NavMesh Baking

- The process of creating a **NavMesh** from the level geometry is called NavMesh Baking.
- The process collects the Render Meshes and **Terrains** of all Game Objects which are marked as *Navigation Static,* and then processes them to create a navigation **mesh** that approximates the walkable surfaces of the level.



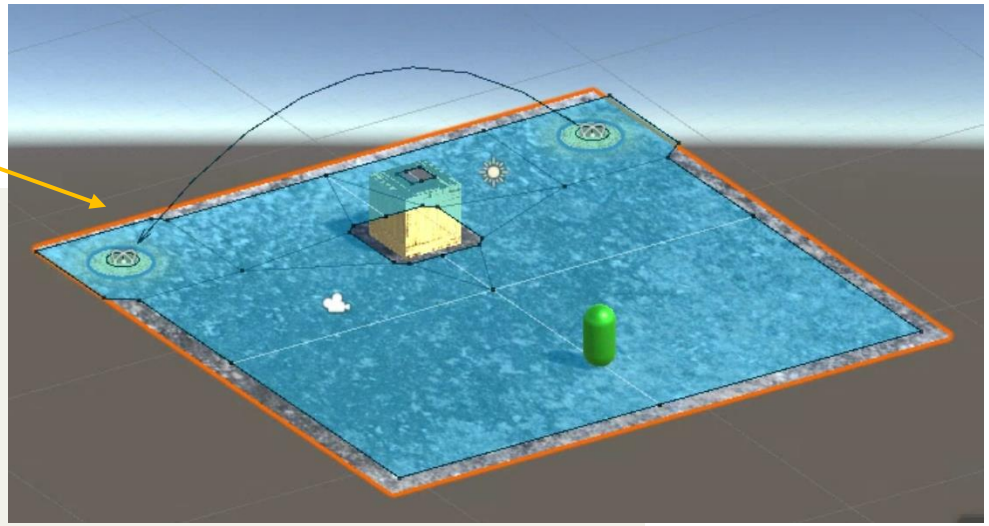
NavMesh Baking

- The process of creating a **NavMesh** from the level geometry is called NavMesh Baking.
- The process collects the Render Meshes and **Terrains** of all Game Objects which are marked as *Navigation Static,* and then processes them to create a navigation **mesh** that approximates the walkable surfaces of the level.



Script Example

The plane is **baked** with NavMesh baking, and the two force fields are linked with **NavMesh link**. Capsule is the **agent.**



```
1  using UnityEngine;
2  using UnityEngine.AI;
3
4  public class MoveToClickPoint : MonoBehaviour
5  {
6          NavMeshAgent agent;
7
8          void Start()
9          {
10             // NaveMesh agent
11             agent = GetComponent<NavMeshAgent>();
12         }
13
14         void Update()
15         {
16             if (Input.GetMouseButtonDown(0))
17             {
18                 // If mouse is pressed draw raycast towards that position and move
19                 RaycastHit hit;
20
21                 if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out hit, 100))
22                 {
23                     agent.destination = hit.point;
24                 }
25             }
26         }
27     }
```

Update()

Draw a raycast in the direction towards the coordinates of the point selected with a mouse click and move.

Scriptable Objects

- A **ScriptableObject** is a data container that you can use to save large amounts of data, independent of class instances. This is useful if your Project has a Prefab that stores unchanging data in attached MonoBehaviour scripts.

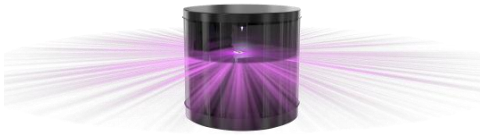
```
using UnityEngine;

[CreateAssetMenu(fileName = "Data", menuName = "ScriptableObjects/SpawnManagerScriptableObject", order = 1)]
public class SpawnManagerScriptableObject : ScriptableObject
{
    public string prefabName;

    public int numberOfPrefabsToCreate;
    public Vector3[] spawnPoints;
}
```

Scriptable Objects

- To use a ScriptableObject, create a script in your application's Assets folder and make it inherit from the ScriptableObject class. You can use the **CreateAssetMenu attribute** to make it easy to create custom assets using your class.

```
using UnityEngine;

public class Spawner : MonoBehaviour
{
    public GameObject entityToSpawn;
    public SpawnManagerScriptableObject spawnManagerValues;
    int instanceNumber = 1;

    void Start()
    {
        SpawnEntities();
    }

    void SpawnEntities()
    {
        int currentSpawnPointIndex = 0;

        for (int i = 0; i < spawnManagerValues.numberOfPrefabsToCreate; i++)
        {
            GameObject currentEntity = Instantiate(entityToSpawn, spawnManagerValues.spawnPoints[currentSpawnPointIndex], Quaternion.identity);
            currentEntity.name = spawnManagerValues.prefabName + instanceNumber;
            currentSpawnPointIndex = (currentSpawnPointIndex + 1) % spawnManagerValues.spawnPoints.Length;
            instanceNumber++;
        }
    }
}
```

EXERCISE: LiDAR system



1. **Bake the NavMesh**

   Bake the environment surface using the *NavMesh* baking tool. Assign the *NavMeshAgent* component to the player, and *NavMeshObstacle* to the other Objects, distinguishing among them selecting the right *LayerMask*.

2. **Detect different categories**

   Change the ray color depending on the object that you hit, detecting the different category objects. You can do it with *Scriptable Object* by assigning a category field defining the class the object belong to or using the *LayerMask*.

3. **Build the LiDAR and store point clouds**

   Increment the number of rays of the raycaster and make it rotate around itself.

   Then, store the acquired points in point cloud files .txt, one for each 360° rotation of your LiDAR scan, storing also the category*, (e.g.,* the *LayerMask*) in addition to 3D coordinates.

4. **(ADVANCED) Mapping environment with LiDAR**

   Make a canvas showing a minimap of the arena and map onto it each object deceted by the LiDAR. Optionally you can also add the player following its movements.

5. **(OPTIONAL) Moving Objects**

   Make fucsia capsules run in the environment with *NavMeshAgent* and store information also regarding the fact that they are movable objects or not, when acquiring.