// Lesson 2 - Physical Raycast

> What you will learn:

- What a raycast is and what it is used for
- Difference between the standard Fudge Raycast and the FudgePhysics Raycast
- How to select an object in the scene with a mouse click

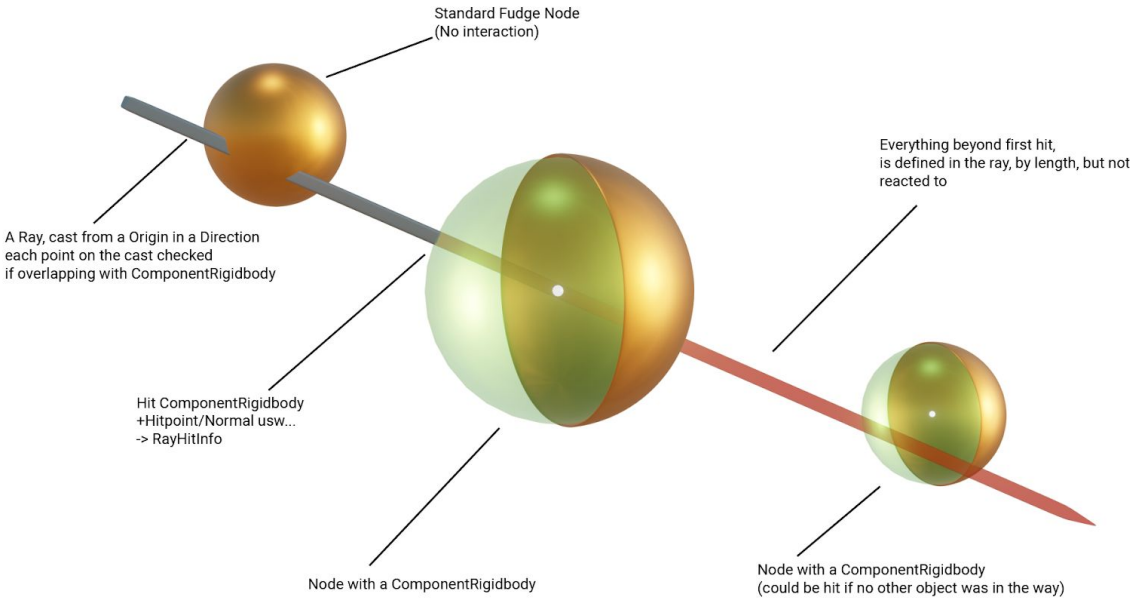
> Requirements Knowledge/Files:

- Knowledge how to setup a (physics) scene in Fudge
- Understanding of the basic physics, learned in lesson 1
- Hint! - You can use the physics boilerplate in the tutorial folder

> TL;DR; - But it's worth to read the long text, to get detailed infos and deeper understanding

- Raycast are used in pretty much every game for selecting, detecting, AI etc.
- Fudge Raycast is pixel perfect but, it's a different system and also informations are different than Physics Raycast, and (in general use Physics Raycast)
- Use f.Physics.raycast(origin, direction,length) to receive a RayHitInfo
- You can filter your cast from, the whole world, a specific group, one body

> Step 1 - What is a raycast and what it is good for



Above is the general visualization of a raycast. A raycast is a mathematical line that has no width, it's checking every point along that line if anything was hit, and returns informations about the hit.

The physical raycast can be thought of like a finger pointing that is stopped on the hit. A ray that is cast into the scene is defined by a origin (a point in 3D world space), a direction (also in 3D space e.g. Vector(0,1,0) for a upwards ray), and a length, how far the ray travels.

Raycasts can be used for a variety of things, their main purpose is to detect hits. Like a mouse click on an object to select it, or a straight shot of a weapon hitting a player. A raycast is a straight line, so it is used for weapons that have no "real" projectile like in most first person shooters. Meaning a projectile would be so fast that it is almost a straight line and not influenced by gravity as much, but it's too fast to be recognized by the physics engine when it would actually collide, since the simulation steps are too small. So you use a raycast that is simple checking if anything is right in front of the weapon in this moment and returns the result, to apply damage.

Raycasts are a basic functionality that is used in nearly every game but rarely noticed. Clicking on any unit in a strategy game, letting ai's detecting enemies and much more is all done by processing raycasts. Fudge received a real "physical" raycast with the integration of physics and those are the focus of this tutorial, but there is also a build-in raycast that can be used.

> Step 2 - Fudge Raycast vs. Physical Raycast

The main difference between the Fudge Raycast and a Physical Raycast is the objects hit, the performance and some informations about the hit itself.

The Fudge raycast works with depth textures.
Meaning it is checking pixels in and their depth in the scene. E.g. you have a red cube on the position (0,0,0) expanding, 1 meter so it's visually possessing some space in the center of your scene. Now we send a Fudge Raycast from (0,0,-5) in a forward direction (0,0,1) with a length of 10 through the scene. The line will hit the cube since one of the pixels is along a 10 meter line that is walking through the scene origin. To achieve that hit, the Fudge Raycast is going through the rendered result image and is checking each pixel and it's scene depth if the depth is on the line. Long story short, and this explanation is already shortened - it's a rather performance heavy process and the result is returning only the hit Fudge Node the distance, depth buffer and the face hit. Good enough for selecting things but not good enough in general.
There lies the advantage of this kind of raycast, because it's pixel perfect you have no collider that might not represent the actual visual of the object. Every shape you can imagine can be hit by this. But it's performance heavy and giving less informations (currently).

So here is the physical raycast, working with "physical" shapes.
Using physics you know the ComponentRigidbody now and their collider. The collider can be hit by a mathematical ray, since the physics engine already calculating "real" physical shapes for the simulation of collision, the extra calculation if a point on a ray is hitting this collider is easier on the performance side and since it's like the ray

is a physical object in the scene - even if it isn't, since it has no shape itself, you receive informations like the normal vector of the hit, distance, hitPoint in 3D World Space and more. The downside is, only objects that possess a ComponentRigidbody can be hit by this raycast instead of every object. This is also an advantage since physical raycasts can ignore objects that are not in the wanted group of objects.

It's recommended to use physics raycast in general for more informations, it's better suited for the general usage within a physics system (raycast is mathematical but in games always part of a physics system). But if you need pixel perfect selection of objects and objects that have no physics component use the Fudge standard Raycast.

> Step 3 - Using the Physics Raycast

We already learned what defines a mathematical ray, and that it is cast into the scene - that's where the name's coming from. So to summarize, it's a straight line in 3D world space, from a starting point to a direction, with a defined length. Think of it like a human walking from A to B without taking detours, but walking into every obstacle in the way, and complaining about what happened.
So the actual ray usage looks like this:

```
f.Physics.raycast(origin, direction, rayLength);
```

We are calling the Physics integration of Fudge to do a raycast for us, giving it the properties and get a RayHitInfo in return.

You also have two options to filter raycasts, in real games you sometimes do not want to hit some objects. E.g. In a shooter a wooden wall that gives no cover. You would only cast objects that are solid + players/enemies. In this case you overload the method with a PHYSICS_GROUP.GroupName.

```
f.Physics.raycast(ray.origin, ray.direction, rayLength,
                  f.PHYSICS_GROUP.GROUP_1);
```

In that case only objects in the specific group can be hit and the closest is returned.

A rare case but sometimes you only want to hit a specific body, this could be in a spy game where you want to check if a specific object is in front of you even through walls/or anything else. For that the integration has the capability to raycast only the specific object.

```
let rigidbody : f.ComponentRigidbody =
node.getComponent(f.ComponentRigidbody);

rigidbody.raycastThisBody(ray.origin, ray.direction, rayLength);
```

We take a specific ComponentRigidbody and cast our ray only onto the collider of this object and see if it it's.

Now, this is how to cast a ray but you want to use the informations of the raycast, so you need the return value of a RayHitInfo, so you should always save your raycast somewhere.

```
let hitInfo: f.RayHitInfo = f.Physics.raycast(ray.origin, ray.direction,
rayLength);
```

Now we can access all of the informations with our hitInfo. The most important is if the ray hit anything and what it it.

```
hitInfo.hit;
hitInfo.rigidbodyComponent;
```

You receive the physics component of the node that is hit so if you want to know the actual node or the name you gave that node use:

```
hitInfo.rigidbodyComponent.getContainer().name
```

> Step 4 - Practical example of using the Physical Raycast

To show a standard use case of a raycast here is a little example. We will select a object with a mouse click and apply some force to it at the exact point our ray hit it, to push it. (Hint! The finished and detailed example is found in the tutorials folder)

Define some physical objects in your scene, the way you learned in Lesson 1. Cubes on a floor is what we use.
As a refresher, we normally have a node, with everything standard attached Component-(Transform, Material, Mesh) and attach a ComponentRigidbody to it.

```
let cmpRigidbody: f.ComponentRigidbody = new f.ComponentRigidbody(_mass,
_physicsType, _colType)
 node.addComponent(cmpRigidbody);
```

With our little scene that consists of multiple cubes on a floor we now have to add a Event on a mouse click. This is not physics specific but just event handling in Fudge.

```
viewPort.activatePointerEvent(f.EVENT_POINTER.DOWN, true);
viewPort.addEventListener(f.EVENT_POINTER.DOWN, hndMouseDown);
```

We tell Fudge to add a event to the viewPort, so our general game, if the mouse is pressed while being in our game window a function is called (handle mouse down) in this case.
So the actual use of the raycast is in hndMouseDown and we build it up like this:

```
function hndMouseDown(_event: f.EventPointer): void {

    let mouse: f.Vector2 = new f.Vector2(_event.pointerX, _event.pointerY);
//Get the mouse position in the html window/client space
    let posProjection: f.Vector2 = viewPort.pointClientToProjection(mouse);
//Convert the mouse position to the projection (ingame space)

//Create a Fudge Mathematical ray. That starts a little in front of the
camera. Using the standard Fudge Ray class just to have a place to save a
origin and direction Vector3 at the same time, you could also have two
separate Vector3 variables
    let ray: f.Ray = new f.Ray(new f.Vector3(-posProjection.x,
                               posProjection.y, 1));

//Re-position the ray start to be at the camera position but offset by the
mouse input. You of course need to store your ComponentCamera in a variable
somewhere beforehand
```

```
    ray.origin.transform(cmpCamera.pivot);
//Create the raycast direction by turning it the way the camera is facing
    ray.direction.transform(cmpCamera.pivot, false);

//The actual raycast
    let hitInfo: f.RayHitInfo = f.Physics.raycast(ray.origin,
                                ray.direction, rayLength);

    if (hitInfo.hit) { //hit is true so something was detected
      //Log whats the name of the hit object
      f.Debug.log(hitInfo.rigidbodyComponent.getContainer().name);

//Push the body at the specific hit point in the direction the ray is coming
from, to push it away from us at the point we hit it, not in the body
center. Notice how you can push it in a very controlled way.
      hitInfo.rigidbodyComponent.applyForceAtPoint(new
                          f.Vector3(ray.direction.x * pushStrength,
                          ray.direction.y * pushStrength,
                          ray.direction.z * pushStrength),
                          hitInfo.hitPoint);
    }
}
```

You'll maybe notice that you can "theoretically" also push the ground, but it won't move if you made it static, in this case you want to have a condition that the hit body either is not in a specific group or does not have a specific name. But that is for you to explore. With the above example you could develop a small soccer goal shooting game where you just click on the ball on a specific point to control how it is shot and define a force by how long the mouse is held down. Just to get your thoughts going.

Hint! - This concludes this tutorial but keep in mind not everything is explained in detail you can explore things better by yourself!