// Lesson 1 - Physics in Fudge Overview and Getting Started

> What you will learn:

- What is the use of physics in games and what are the basic features
- How to use them within Fudge
- Manipulating physical properties to simulate things

> Requirements Knowledge/Files:

- Knowledge how to setup a scene in Fudge
- Knowledge of the basic Component System of Fudge
- 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

- Physics is nearly every complex game, and a great addition to games
- You need to include OimoPhysics.js included in your HTML you find it in the Physics Folder of Fudge
- You need to have f.Physics.start(yourRootNode); called before your game loop starts
- Physics provides 4 new big features to Fudge Collision/Forces, Raycast, Physical Events, Joints
- There are new Components, for physics, ComponentRigidbody and ComponentJoint_JointTypeName (e.g. ComponentJoint_Prismatic)
- ComponentRigidbody means a solid body, not squishable, with a weight, a collider and a physical reaction type
- ComponentRigidbody has a lot of properties that change the simulated behaviour like friction/restitution that can be accessed.
- Physical objects are moved by forces and impulses, resulting in velocities. Positions/Rotations need to be accessed through physics not transform.

> Step 1 - Physics their use within games an the basic features of a physics engine

Did you ever fell in a game? Or walked face forward into a wall? Shot an enemy or a goal in a soccer game? Probably and there you have it, the use of physics in games.

Physics in games pretty much is used to have a believable representation of things happening on their own like they do in the real world. Real can be understood as a synonym for physics are applied. In old games physics where often not so sophisticated, because they simulated only small parts of a physics engine, often manually.
Falling for example is nothing else than adding a accelerated velocity to the translation of an object each frame to make it look like it's falling. But applying this concept to each object and have it consider weight, air resistance of anything else is a hassle. So over time concepts that apply in the real world where collected and visualized in physics engines, that only need a few informations from the game developer to simulate "the real physical world" on it's own.
Fudge has recently integrated OimoPhysics as their physics engine of choice. But no worries you do not have to deal with anything else but structures and concepts you already know from Fudge because integration means they work together behind the curtain but you only have to deal with the Fudge part, mostly.

The basic features of a physics engine are those of so called rigid bodies.
A concept that we know as bodies that are not squishable in any shape or form. Take a stone for example it's solid, heavy, rough and you can press your finger on it but it does not change its properties. Now the opposite of this concept are the so called soft bodies, like a sponge, fluids, cloth and such. These are a advanced form of physics and are not part of a standard physics engine. But you might come across those when using other engines in the future.

So in Fudge you work with rigid bodies, and since Fudge is using a component system with different features attached as components to a NODE, the new main component is the ComponentRigidbody.

Which results in the first feature of having a physical solid body that is possessing different properties and is reacting to other bodies according to basic physical rules. Like being in the way of objects, so you have a collision. These rules allow a behaviour that is expected and mostly always the same if the same properties are still in place. - Lesson 1 (this lesson)

Second feature is those of raycasts a possibility to interact with physical objects without being one itself. You will learn about that in Lesson 2. Simply put, a ray in a scene detecting objects - Lesson 2.

Third feature is the implementation of physical events. The topic of Lesson 3. Simplified two rigidbodies meet in the simulation and the developer has the option to let something happen - Lesson 3.

Fourth feature is the introduction of joints. A way to connect physical bodies by attaching them with mechanical force. For example you take a wood board and screw it to the wall. They are now unable to act independent from each other but depending on how hard you screwed them the board can still rotate. This feature is needed because you might have used Scene Hierarchy or so called Transform Hierarchy to parent a Node to another. But here comes physics, physics ignores these structures. Because real physical objects have no parents, they are either held by force, think of laying a book on another book on a table, they do not fall because they are held, obviously. Or held by screws or nails connecting mechanical parts, so called joints - Lesson 4.

And lastly not a feature of physics but a feature that is needed because of it, the debugging, visualization of physics to have a way to find problems within your simulation - Lesson 5.

So now we know the goals of using a physics engine. But we haven't used it yet.

> Step 2 - Use Physics Within Fudge

OimoPhysics is integrated with Fudge but it does not come with it, so projects without physics do not have to load a whole engine that they don't need. So to use it generally you have to include it in your HTML file.

```
<script src="../../../OimoPhysics.js"></script>
<script src="../../../../Core/Build/FudgeCore.js"></script>
```

Similar to the FudgeCore just have a reference in the head of the html file, your relative path might look different. And also you have to use the OimoPhysics.js that is within the Physics Folder of Fudge, because only this version of the engine is useable with Fudge.

Step 2.1

Great, now Fudge knows who to talk to, but we still have to cover what it is saying, how and why.

To get things out of the way, before our game starts we have to tell the physics engine what is our scene root. The scene root is a concept you should know. It's the first object in the hierarchy. The physics engine is using it to make sure the starting properties of rigidbodies are correct.
So usually you start your game by having a update function, a loop, that every frame is recalculating and redrawing the scene. Now before you do this you have to start the physics engine.

```
f.Physics.start(hierarchy); //hierarchy is your base/root node in the scene

f.Loop.addEventListener(f.EVENT.LOOP_FRAME, update); //Tell the game loop to
call the update function on each frame
f.Loop.start(); //Start the game loop
```

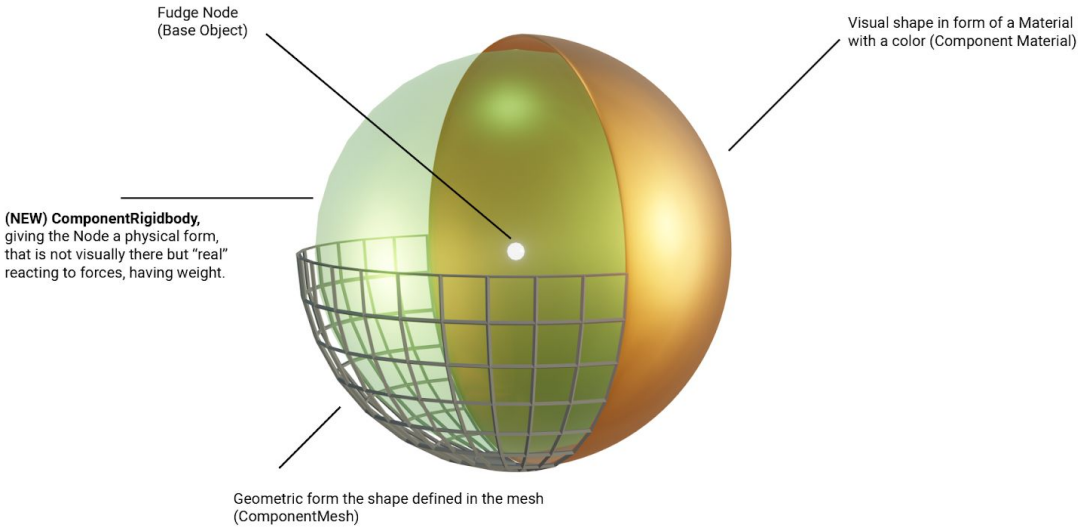
By the way, all of these physics tutorials expect you to have f = FudgeCore defined on the top of your typescript file, to reference Fudge quickly. Typically the f is often a ƒ, the so called florin sign. when you are using other Fudge tutorials, by Professor Dell'Oro, which is alt + 0131 on windows.

Now you learned that the physics integration is accessed by f.Physics, the main physics class which is simulating and keeping track of rigidbodies. Most of the times

you do not need to interact with it further but you can for example access some settings by f.Physics.settings.

Step 2.2

The concept of a rigidbody. The rigidbody is a new component and therefore like any other component. To give more perspective on the concept, look at the visualization below, the new component of the node is now the green orb.



To use this component you only need to attach it like any other component but with other parameters. And it comes with properties that can change the behaviour of it in the physics simulation. But you still have other components like the visual material, the visual shape in form of a mesh, the transform for positioning and such, and naturally the node itself, that is holding all the components.

Converting a standard node to a physics controlled node is therefore done by adding the new component.

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

node.addComponent(cmpRigidbody);
```

A rigidbody is basically defined by having weight, a mass, a type of reaction to physics simulation, a collision-shape (called a collider). It's prefered to speak of bodies when using physics instead of objects or nodes. So in this tutorials objects with physics will always just be called rigidbodies or bodies.
While having a weight is clear, it's just the heaviness in kilogram. The physics type is a more complex subject. There are three different types of physical interaction that a body can have, that can be found within f.PHYSICS_TYPE:

- Static: Immovable, but interacting with other bodies, mostly used for things like obstacles, walls that are fixed. They do still have a friction and restitution that they apply to other bodies but they themselves won't react

- Dynamic: Standard physical object, interacting with everything in the defined physical way. They naturally only react to physics, changes in transform after the start of the game will be overwritten. All transformations are in world space

- Kinematic: Is able to interact with dynamic physical objects, but is controlled by transform/animation. However since they can't be pushed by physics, but only push themselves, they will move through objects that won't move themselves e.g. static objects. This problem can be tackled by the usage of raycast for example.

You also need to define a collider which represents the shape the object has in the physical world, the collider can be different than the visual representation. You can have a sphere that is colliding like a cube. There are predefined colliders:

```
CUBE, SPHERE, CAPSULE, CYLINDER, CONE, PYRAMID
```

And also the convex collider, which is a special case. The shape is dependant on either the scale of the node in world space or on the pivot you defined within the ComponentRigidbody. The convex collider is a type of collider that has no predefined

shape but takes fed in vertices to calculate a shape. With this you can shape things like gemstones with odd faces. But convex means it's a closed of shape, with a mathematical genus of 0, meaning there are no holes in it. A torus for example is not a mathematical stable shape and has therefore a genus of 1, the obvious hole in the middle.

A full node creation function to have every basic component attached can look like this: (This will be referred to and used in all later tutorials)

```
function createCompleteNode(_name: string, _material: f.Material,
_mesh: f.Mesh, _mass: number, _physicsType: f.PHYSICS_TYPE,
_group: f.PHYSICS_GROUP = f.PHYSICS_GROUP.DEFAULT, _colType:
f.COLLIDER_TYPE = f.COLLIDER_TYPE.CUBE): f.Node {

//Standard Fudge Node Creation
    let node: f.Node  = new f.Node(_name);

//Creating a mesh for the node
    let cmpMesh: f.ComponentMesh = new f.ComponentMesh(_mesh);

//Creating a material for the node
    let cmpMaterial: f.ComponentMaterial = new
f.ComponentMaterial(_material);

//Transform holding position/rotation/scaling of the node
    let cmpTransform: f.ComponentTransform = new
f.ComponentTransform();

//Adding a physical body component to use physics
    let cmpRigidbody: f.ComponentRigidbody = new
f.ComponentRigidbody(_mass, _physicsType, _colType, _group);

    node.addComponent(cmpMesh);
    node.addComponent(cmpMaterial);
    node.addComponent(cmpTransform);
    node.addComponent(cmpRigidbody); // <-- best practice to add
physics component last
    return node;
  }
```

With this information you can basically use your bodies to simulate physics and influence game logic later. But there is more, each body has properties that can be influence to make it feel more realistic and different or serve a specific need.

> Step 3 - Physical Properties Usage And Manipulation

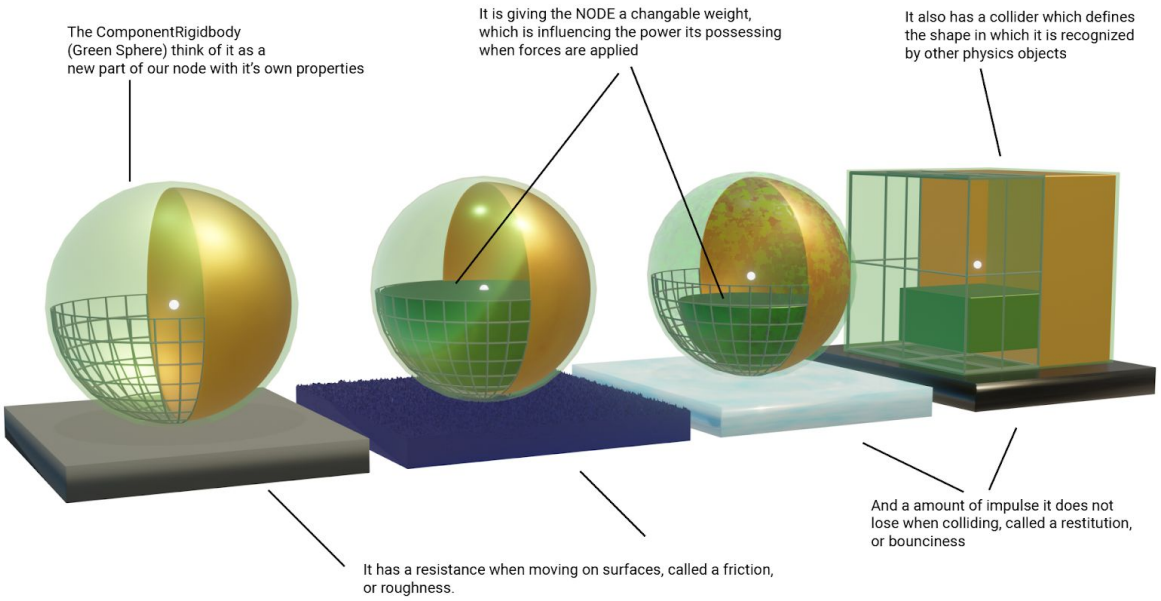
To manipulate a property just talk to the rigidbody and change the property.

```
cubeNode.getComponent(f.ComponentRigidbody).friction = 0.02;
cubeNode.getComponent(f.ComponentRigidbody).propertyName = value;
```

Every property can be accessed as a part of the rigidbody class. The most common are the friction and restitution. Or in layman's terms, how much it's sliding and bouncing. Keep in mind that physics always happen between two bodies, even the air around the body is a body in itself. You'll notice that when changing the linearDamping and angularDamping, simplified the air resistance of your rigidbody. Since there are always 2 bodies involved the friction is applied between those. Like ice slides perfectly on a surface like clean metal, it's nearly stuck on carpet. So when you want a ball to still bounce of a floor with very low restitution you need to set the restitution of your ball absurdly high.

When designing Nodes and giving them a material that might visualize a metal you should also learn about the physical properties of metal, like it's heavy weight, low restitution, but if it's not rough, relatively low friction.

The below visualization might make it clearer of how a ComponentRigidbody is changeable and how important that can be in games/simulations.



There are also properties

Now that we have defined our bodies, and understand some properties we have to think about influencing their behaviour at runtime.

> Step 4 - Forces and Impulse, Positions/Rotations, Physical Bodies in Realtime

Now when you created your first body and it's dynamic it will fall, that's the basic functionality of all physics that we know of, everything is pushed towards the earth with 9,81 m/s. Also known as gravity (which you can also change within f.Physics.world.setGravity(value); or for each body manually). This is not a number that is just added, it's a acceleration-force on our bodies, and it's constant, all around us, that's why we do not notice it.

There is also another form of force, it's instantaneous, when you hit a ball with your foot. Or a glass hits the ground. It's technically a force but since it functions differently it's called an impulse!

So these are the different type we can apply to our bodies to influence their behaviour outside of what they do to each other within the simulation.

```
//A force applied to a body, pushing it
```

```
cubeNode.getComponent(f.ComponentRigidbody).applyForce(new
f.Vector3(0, 750, 0));

// An impulse applied to a body, hitting it, so the amount of
force in newton is smaller, since it's a force applied in an
instant
cubeNode.getComponent(f.ComponentRigidbody).applyImpulseAtPoint(ne
w f.Vector3(0, 15, 0));
```

A great feature of a physics engine is, we can also apply forces and impulses at points in the world and the physics engine calculates how much of the force is hitting the body and where and what are the effects of it. While we can't push the position of a cube for example only on the top left corner, with transform, without thinking about the complex things happening, the physics can rotate it exactly like someone touched in on that spot. And all of that with things like collision and friction between other bodies still applied.

```
cubeNode.getComponent(f.ComponentRigidbody).applyForceAtPoint(new
f.Vector3(0, 750, 0), new f.Vector3(0,5,0));
```

This way you might simulate explosions, earthquake or whatever you like. Impulses are always on a point but if no point is given, the point is the center of the body.
Applying Forces, Impulses and Torques (rotational forces), result in a velocity the body has. This can also be influenced if you don't want to push a body with a force and calculate how much force would be needed to reach the velocity you want, instead just manipulate velocities directly.

```
cubeNode.getComponent(f.ComponentRigidbody).setVelocity(new
f.Vector3(0,2,0));

cubeNode.getComponent(f.ComponentRigidbody).setAngularVelocity(new
f.Vector3(5,0,0));
```

But keep in mind there is a linear velocity, often only called velocity that is implying the velocity the body has just flying in a straight line without any rotation. and a

angular velocity how much the body is spinning. The speed, which is often used is the magnitude of the velocity vector. When moving only in one direction speed equals the velocity in that direction.

Dynamic physical bodies do not listen to the transform hierarchy, this is because in physics there is no hierarchy and no transform. To move a physical object without pushing it but directly from one place to another - teleport, there is the possibility of directly accessing the position/rotation of the body in the physics simulation.

```
cubeNode.getComponent(f.ComponentRigidbody).setPosition(new
f.Vector3(x,y,z));
```

Don't forget that, it's a common problem that bodies need to teleport and the developer is not using physics and afterwards they wonder why it's not moving.

Forces / Impulses are applied, and not added, that's because you do not directly "add" movement to the body. It's just a part added to a formula that is then calculated to create a movement. Think of simplified basic physics. f = m*a, you're object gets a acceleration out of the force considering its weight. To have the actual velocity change within one frame you need to think further, $v = a \cdot t + v_0$ and at this point we know why we have a physics engine.

The physics simulation is adding a huge amount of features to Fudge and is working within it's structures but there are rules to physics that can't be broken or else a somewhat realistic simulation would not be possible.

Hint! - This concludes this tutorial but keep in mind not everything is explained in detail you can explore things better by yourself! There are examples within the tutorial folder to follow along and get a grip on physics.