// Lesson 5 - Debugging Physics and change Settings

> What you will learn:

- How to use the debugging capabilities of FudgePhysics
- Using the given informations to improve your scene

> Requirements Knowledge/Files:

- Any Fudge-Scene that is using “Physics”

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

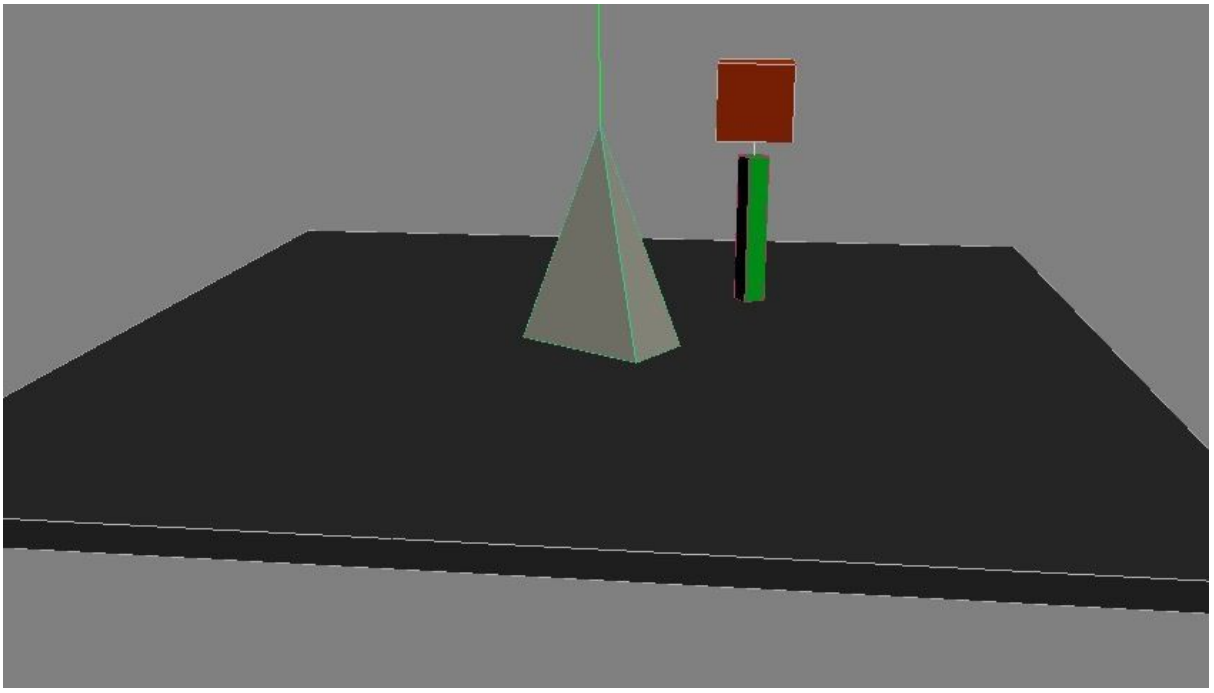
- Use Physics Debug Draw by activating it in the physics settings, with Physics.settings.debugDraw = true
- There are different debug modes depending if you want to know something specific, but the default tells most of the things
- If you can’t see objects use the debug mode that draws only physical objects as wireframes
- There are a few physics settings and to increase accuracy of the physics simulation use Physics.world.setSolverIterations(number), obviously with a performance cost

> Step 1 - Activate Debugging Draw

Each frame Fudge is drawing every node with their defined scene onto the html canvas, the standard rendering process. Physics are complicated and the numbers often don’t tell the whole story behind calculations.
So we can tell the physics integration to inject some physics information into the render process and visualize our simulation.

```
f.Physics.settings.debugDraw = true;
```

This does call to our Fudge Class/Engine with f (you should have defined f = FudgeCore) and we dive deep into the Physics integration and change a settings property in this case the debugDraw, simply set to true.
This is changing our scene from a game to more of a editor/developing stage. Where we get informations about what's happening to our physic objects, so remember to set it back to false when shipping your game.



What you now see as default is a wireframe around all physic objects and if you used joints in your scene some lines. You are in the default debugMode that is drawing the collider of your objects and axis/anchors/constraints of joints. That's why it's called the JOINTS_AND_COLLIDER mode. We learn more about different modes and the informations you receive from this debugging, in the next step.

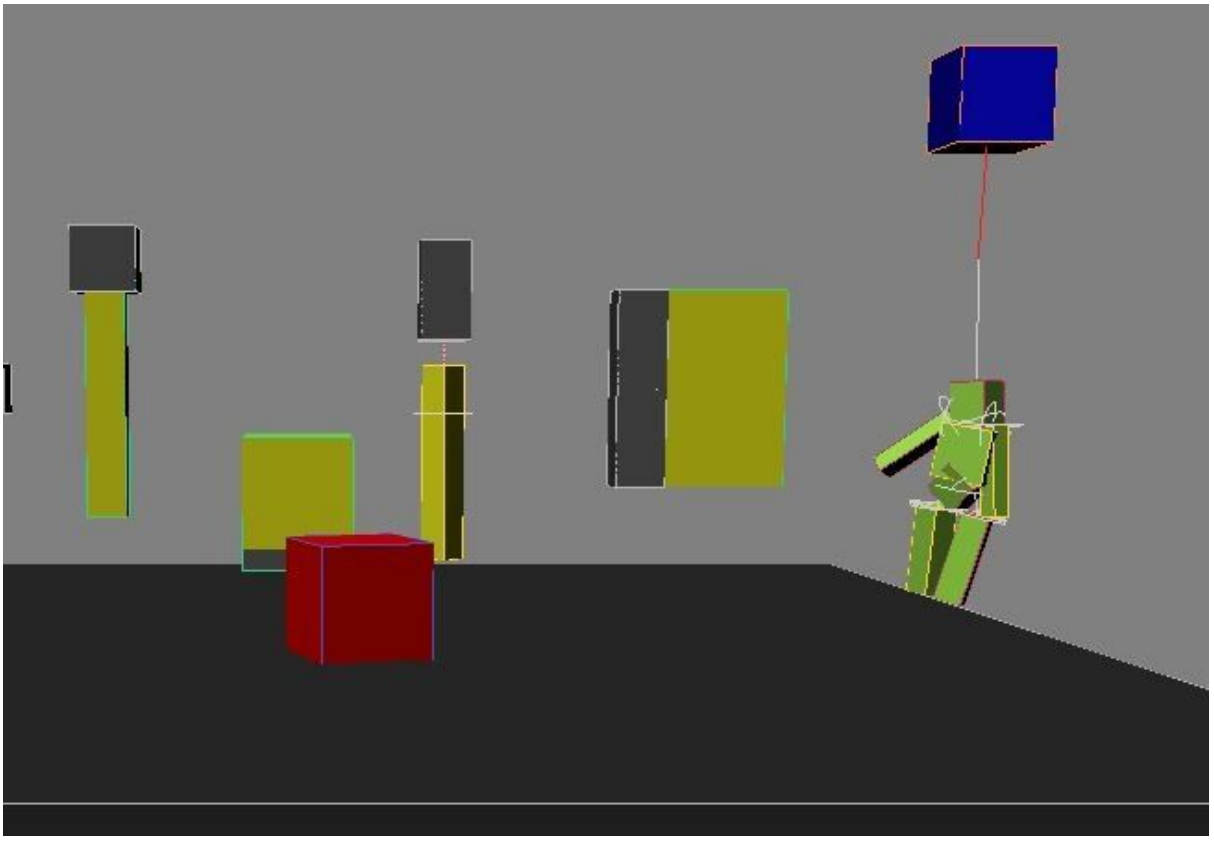
> Step 2 - Debug Modes

When the debugDraw is active it is possible to change what informations are drawn. They are grouped in 5 different modes for specific use cases. We learned in step 1 that the default is the JOINTS_AND_COLLIDER mode that is showing informations

about the shape and size of your colliders and things like limits and directions of joints. But you will notice that the wireframes of your objects change color after a while or when hitting something. This coloration is indicating the status of the collider.

White, means the collider is static, so it's immovable. While blue means dynamic but sleeping. Yellow is dynamic and currently awake (so it's still moving or not standing still long enough to sleep). Orange tells you that this body is kinematic, so physics do react but it's moved by the ComponentTransform. There can be combinations of the colors that will indicate a inbetween status, so objects can be red/violet/orange.

Joints are generally indicated by white lines. These lines are relatively self explanatory a line is visible that is indicating the movement limits of a translation and a joint in general. Circles/Arc's indicate the movement limits of the rotation. Springs are indicated by white lines and if they are stretched the stretched part is drawn as a red line.

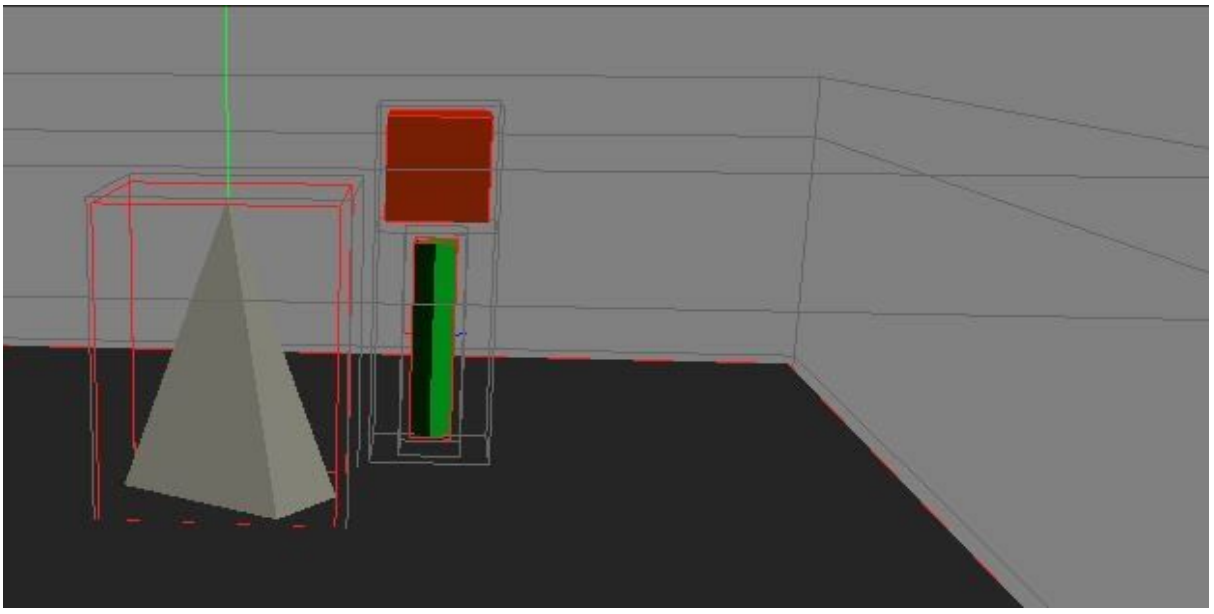


Now there are 4 more modes and you can access them again through the Physics.settings.

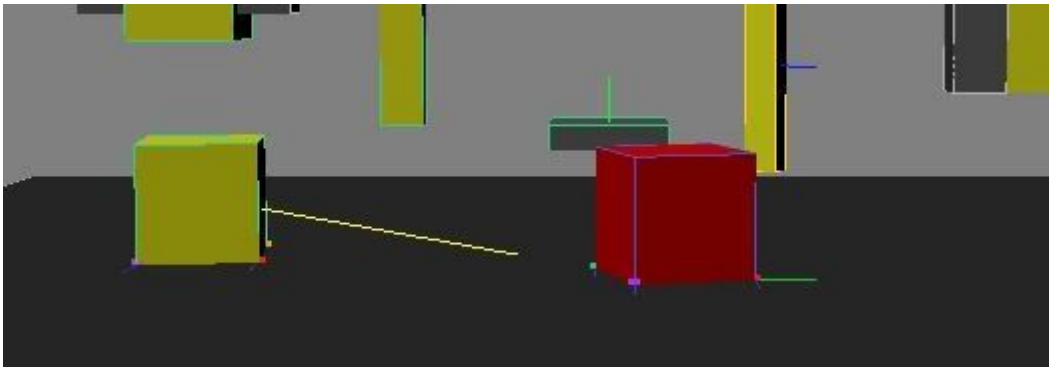
```
f.Physics.settings.debugMode = f.PHYSICS_DEBUGMODE.COLLIDERS
```

All debug modes can be found in the enum PHYSICS_DEBUGMODE the above is the mode that is focusing on colliders. This gives you even more detailed informations about your bodies colliding capabilities. In Form of the direction each body is facing, which is helpful to check that rotations within joints are correctly applied. This is mostly a stripped down mode of the default mode for better oversight.
Now the last 3 modes are:

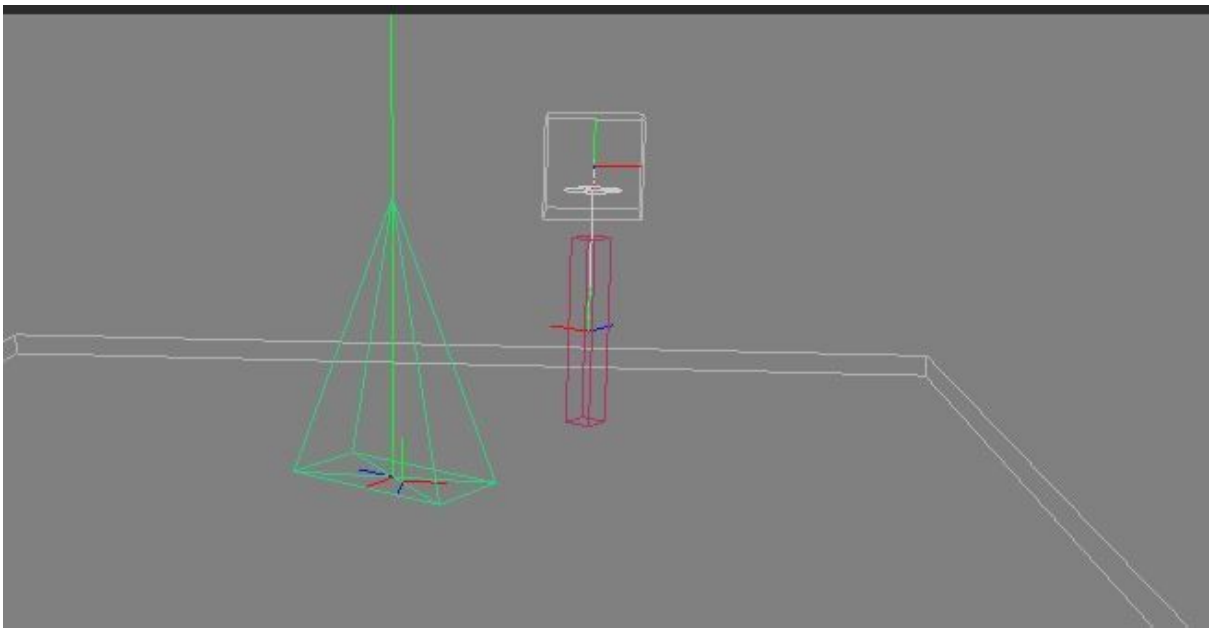
BOUNDING BOXES - This is visualizing in red the space that the body is taking in form of a box. This indicates the space that is occupied in the first calculation of your physics simulation. When the calculation steps are low and your objects are fast bounding boxes that are much more sizeable than the actual collider can detect hits even if they aren't actually colliding. So a bounding box that is closer to the visual representation of the object or the collider is better, but you can't do much about it in most cases, as you can see on a pyramid, it's better in this case to increase the amount of calculations the physics engine, we will see how to do that later.
The grey boxes show informations about the clusters that are defined to calculate the physics, the less the better, and the bigger the better, because much of your scene can be calculated in a bigger step. The drawn information is also called the broadphase-volume-hierarchy, because it shows the steps the physics engine takes from biggest broadphase-volume to the smallest.



CONTACTS - Pretty self explanatory, the informations about actual touching collisions is drawn. Every point where an object is touching is drawn as a small point and consists of the 3 direction lines (green = Y, red = X, blue = Z). You will also notice yellow lines from touching bodies which indicate that the bodies are engaged in a collision and it's drawing a line between their centers.



PHYSIC_OBJECTS_ONLY - Will **only** draw nodes that have a ComponentRigidbody and will draw most of the informations of the default mode + their directions. This mode is useful because often the mesh or non-physics objects are in front of your physics objects, hiding useful informations, so you can either not see things like joint limits or other informations.



Hint! - Raycasts are drawn in every mode, but they are only drawn in the frame they are happening, so click raycasts are often only drawn for a single frame. To debug those it's often needed to move them momentarily in a continuous function like update.

> Step 3 - Physics Settings

We used the settings already to activate the debugDraw, but there are many more settings. A standard user normally has no need for changing most of the settings but it is good to know they are there.
To setup defaults or change things at runtime. All but one can be accessed by.

```
f.Physics.settings.SETTINGNAME = VALUE;
```

Some of them are:

```
.disableSleeping = true;
```

A body is sleeping when no new physics are happening to it for a while. To increase performance bodies that are unused sleep but when they are used again they need to be woken up which can induce errors. So in some cases it can help to disable sleeping but you will see a drastic decrease in performance on large scenes without

sleeping. A sphere for example can sometimes look like it's rolling and then freezing instead of rolling out until it stops because the remaining rotation is so small that the body is going to sleep early, in this case it is good to disable sleeping for better visual quality. You can also change the sleepingThresholds in the settings.

```
.defaultRestitution = 0.5;
```

Instead of setting every rigidbody manually when you want a whole scene or most of it to have different restitutions/friction you can set it in the settings as a new default.

So the **one** setting that is important but is not in the settings is the quality of the actual physics simulation. This setting is part of the actual physics world.

```
f.Physics.world.setSolverIterations(100);
f.Physics.world.getSolverIterations();
```

This will change the amount of iterations the physics engine is using to calculate correct physical events happening but also influence performance. Fudge has integrated OimoPhysics as you know by now, and this means you can crank up the number higher than with other physic engines since Oimo is using a correction algorithm in most cases instead of actual iterations. Default is ~10 iterations, and that is enough most of the times. But for serious work with joints or complex scenes you often need much more, 50-100 is a good starting point, in other engines you should rarely go higher than 20.

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