Tutorial for a kinematic FPS game

Gokudomatic

January 31, 2015
v1.2

Revisions

1.0	Original draft
1.1	Fixed elevator bug
1.2	Small grammar corrections

Contents

1 Introduction

1.1 Preface

Hi folks! Today we're going to build a basic FPS (First Person Shooter) game with Godot. The highlights of this tutorial are to set up a 3d project, to import 3d models and to implement a script to make the character move like in old FPS games like Doom[9], Duke Nukem or Half Life.

I decided to write this tutorial while I was working on another Godot project which requires characters able to move in a 3d world. I noticed that Godot was not offering out of the box such features and the existing similar demos (3d platformer, kinematic char and FPS Test) do not provide exactly a ready to use solution that behaves like in the games mentioned above. I played then for a while, trying to understand little by little the code of the existing demos and figure out why they behave strangely in some situations, like slopes and stairs. Finally I got a satisfying implementation of a solution using kinematics and a couple of others concepts. And then I decided to share this work as a tutorial. I hope it will at the same time help beginners to understand how to build a 3d game with self made assets.
It won't cover however the basic stuffs from Godot, like the concept of nodes, the scripting language GDScript or the user interface. If you are interested in those topics, I wrote a 2d demo[8] with a documentation that explains how nodes work in Godot and their capabilities.

I'm aware that as any tutorial, this one will eventually become outdated. However I'll try as much as possible to explain the maths behind, which is the most important part of this tutorial. I hope that even if Godot changes radically, you can still be able to rewrite easily the code. In fact, if you can even implement it in another game engine (Unity for instance), this document reached its objective.

What it covers Within this tutorial, you'll learn to:

- Create a 3d scene.
- Prepare a 3d model in Blender and export it.
- Import 3d models in Godot.
- Implement a kinematic based First Person View character.
- Create lifts and ladders.
- Create actionable doors and buttons.

What it doesn't cover This tutorial does not intend to explain every part of an FPS game, as it would be too much to explain in one tutorial. And that would be too specific to one kind of gameplay anyway. What won't be covered, among lot of others stuffs, are:

- Sounds

- Lights and shadows
- Enemies, NPC and bullets
- HUD and menu
- Death by falling or being crushed
- How to make a 3d model. I briefly cover how to build a map in inkscape and blender, but I assume you know how to use those softwares. It will mostly be hints and tips specific to godot.

1.2 Target audience

This document is destined to advanced beginners. A basic knowledge in Godot (IDE, nodes, script language) and in programming, including UML, is mandatory to properly understand this tutorial.
Knowledge in Sketchup and Blender is also helpful, but you can download the 3d model from the project[1] if you want to skip the modelling part.

About math a good understanding of vectors is required. It is advised to get a grasp of the basics of quaternions too since we're going to use them a bit. You can find tutorials about those points in the wiki of Godot [3].

1.3 Requirements

For this tutorial, you need to install:

- Godot v1.1, build from github source or later version[2]
- Blender 2.7 or later [5]
- Better Collada exporter plugin for Blender provided with Godot. You'll have to install it in the Blender folder.

I will also use Sketchup [6] and textures from Freedoom[10], but it is optional. If you use Sketchup, you'll need a script[7] to export your model to STL format.

This tutorial was written in Xubuntu 14.10 with a night build from Godot and Blender 2.72b, as well as Sketchup 8 through Wine. The Better Collada exporter is however from the stable version 1.0 of Godot, since the night build version when I was writing this document had a bug.

1.4 How to read

This document should be read step by step. If you don't want to use sketchup, you can skip chapter 3. And if you have already a collada map or if you want to use the map of this tutorial, you can also skip chapter 4.

1.5 Terminology

The steps of the tutorial are displayed like this:

1) Step one

2) Step two

3) Step three

GDScripting code:

```
# comment
var v=Vector3(1,0.5,10)
print("vector: ",v)
func _process(delta):
  pass
```

References to source code(variables, node names, classes) or Godot API are in *italic*.

Files, folders and paths are ”quoted”. The paths will always be relative paths.

Warnings and important information:

Important!

This is an important note.

Hints and tips:

Hint

This is a tip.

Keyboard hot-keys will be displayed like this : **[Ctrl-A]**
If, like often in Blender, a sequence of keys must be pressed, it will be displayed like this : **[A,B,C,D]**

1.6 About the author

I’m essentially a programmer with a few experience in 2d and 3d graphics. I have some personal experience in game programming, mostly in Java and Delphi, yet all my projects were for personal training and didn’t reach any public status.
I played with Godot almost since it was open-sourced, but because my knowledge in 3d math are weak, I was mostly sticking with the 2d part of the engine.

2 Project folder structure

Let's start!

1) Create a folder with 2 sub-folders named "assets" and "source".

Hint

The folder "assets" is for all your media files. In this case, it will be sketchup and blender files, as well as textures. But it would be for audio files too if your game uses some, which is most probably the case.
The folder "source" the Godot project itself and all its source code.
By separating the assets and source, you won't distribute the original files of your assets when you export your project.

2) Open Godot and create a new project in the folder "source".

3) In the "source" folder, create a folder named "game_assets".

Hint

This sub-folder will contain all assets that need to be distributed with the game.

4) Copy all the textures you need in both "assets" and "game_assets" folders.

Hint

For big projects, it is strongly recommended to use a VCS (Version Control System) for at least the "source" folder.

Note As a general rule, save regularly. No matter which tool you're using, don't forget to save in case of a crash.

3 Build the map in Sketchup

We're going here to draw everything specific to the map, including elevators and ladders. You are free to draw any kind of map, but to properly show what the demo can do, it's better that it includes at least:

- 1 elevator
- 1 button
- 1 ladder

- 1 slope of 30° or less (where you can stand on)
- 1 slope steeper than 45° (where you'll slide)

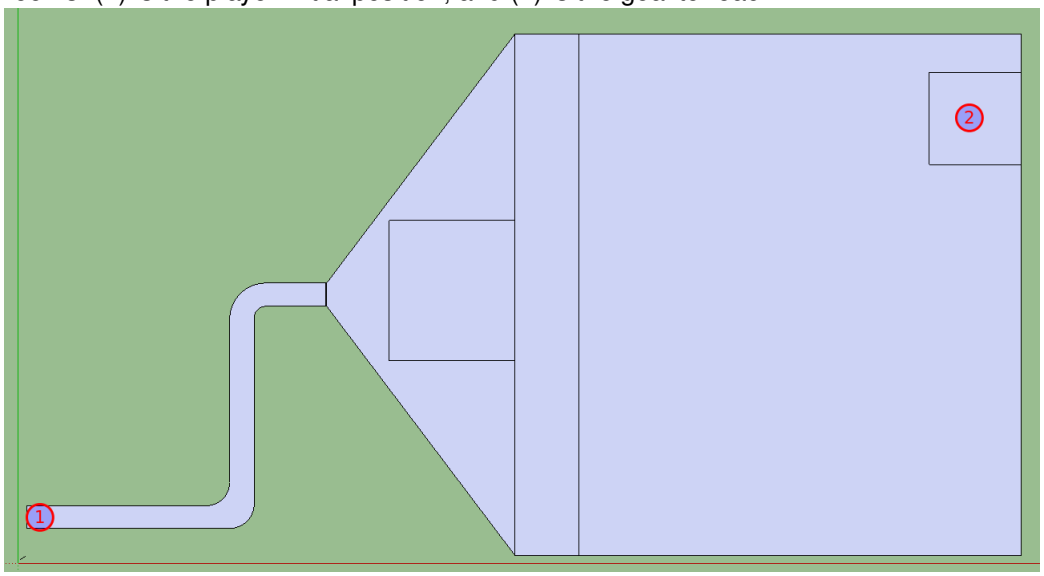
I'll add a door later in Blender because it is more convenient.

Hint

Elevators, buttons and others non-static objects can be drawn here. But for larger projects, you might prefer to make reusable models, since buttons and doors need script. In this case, it is better to add them later in Godot.

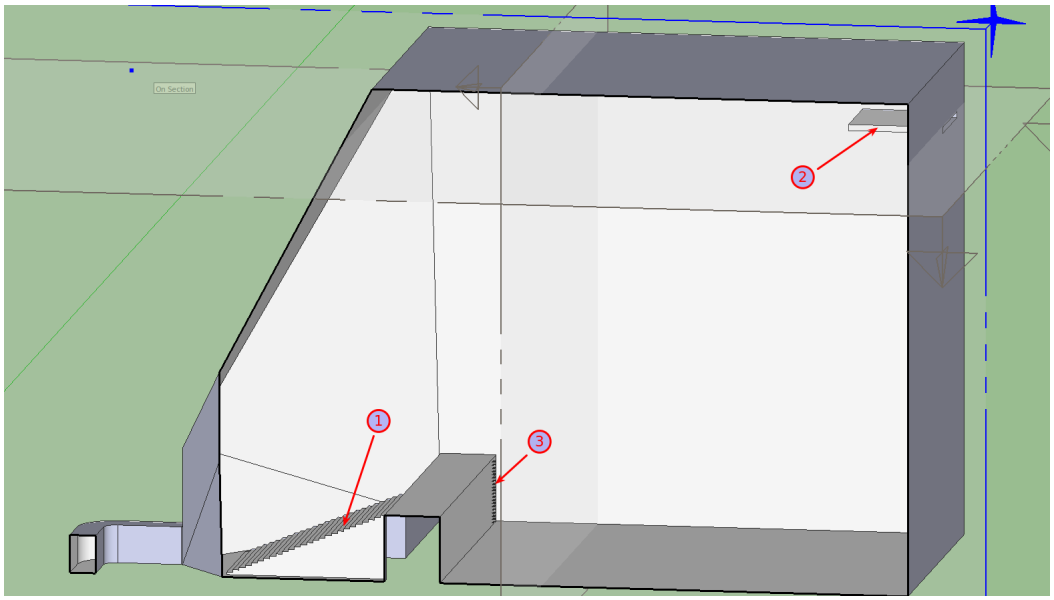
3.1 Draw the map

5) Open a new project in Sketchup and set the view on top. Then draw the rooms. (1) is the player initial position, and (2) is the goal to reach.

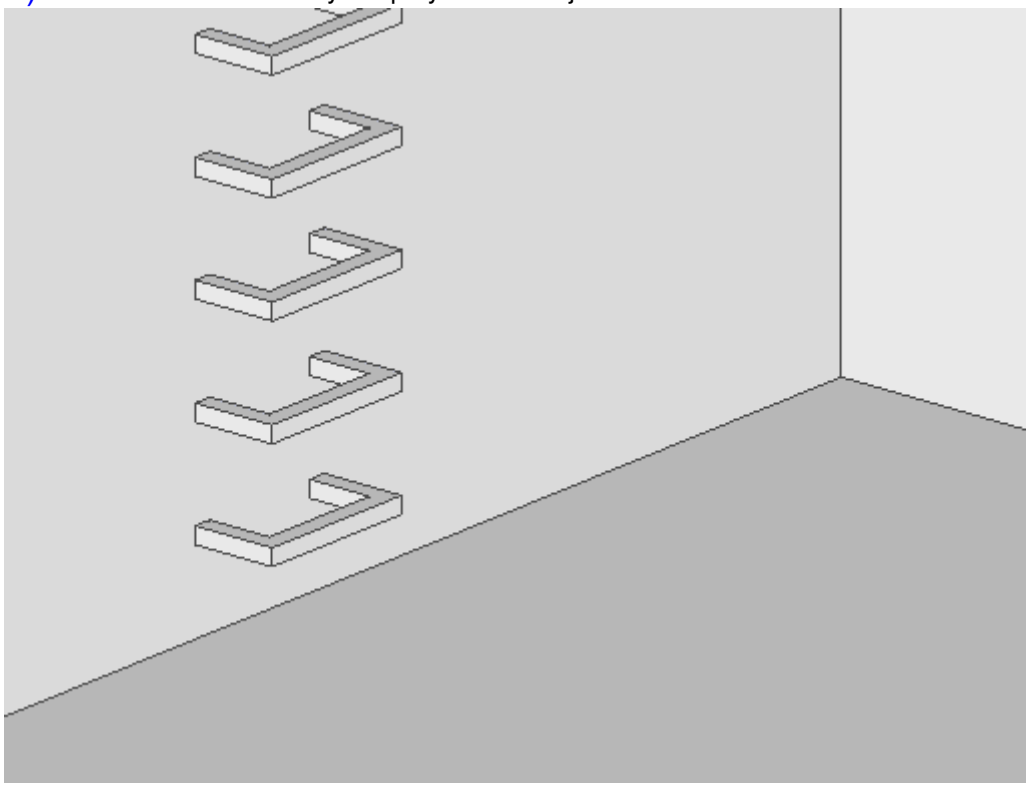


6) Extrude the rooms to make walls and ceilings.

7) Create stairs (1), extrude from the wall a platform for the goal (2), and make a ladder (3).

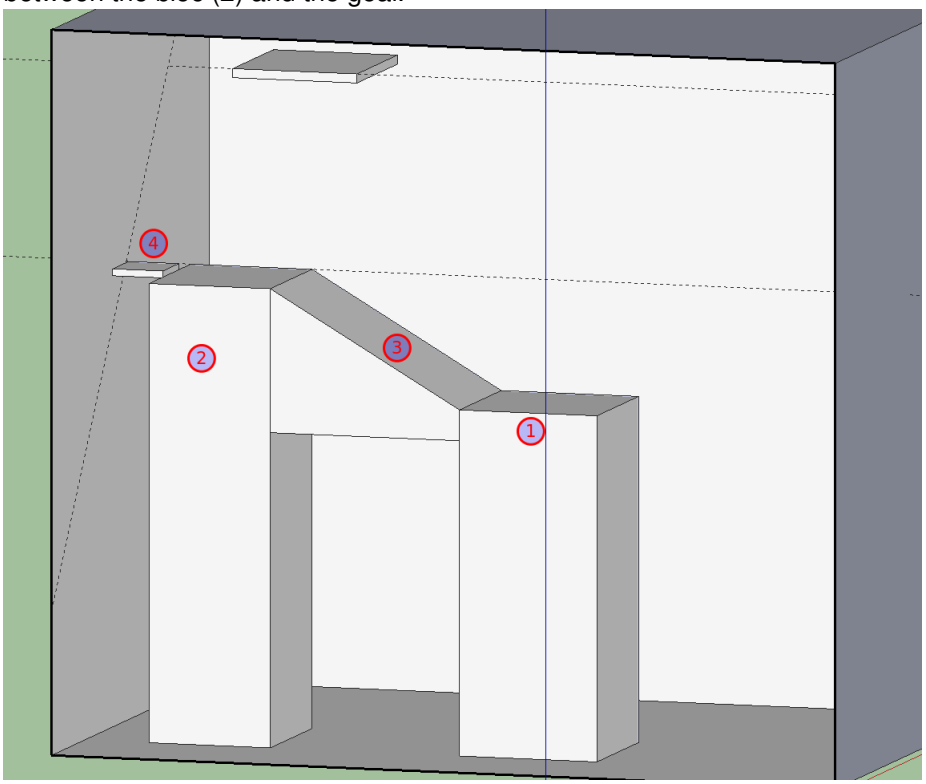


8) The ladder can have any shape you wish. It just needs to be vertical.

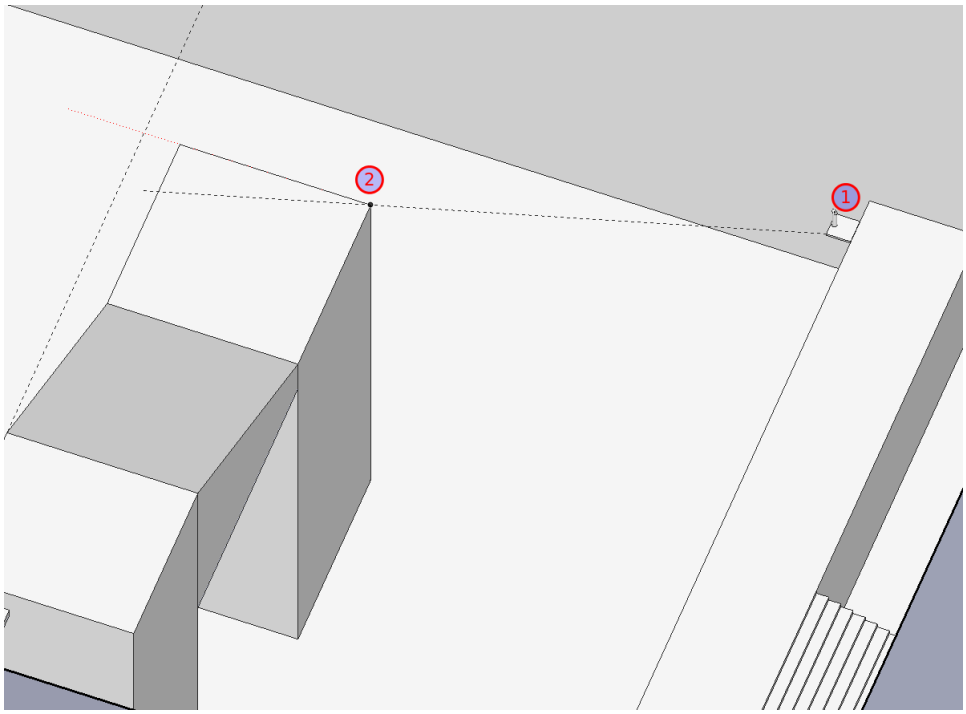


9) Create 2 blocs in the middle of the room (1) & (2), and a bridge between

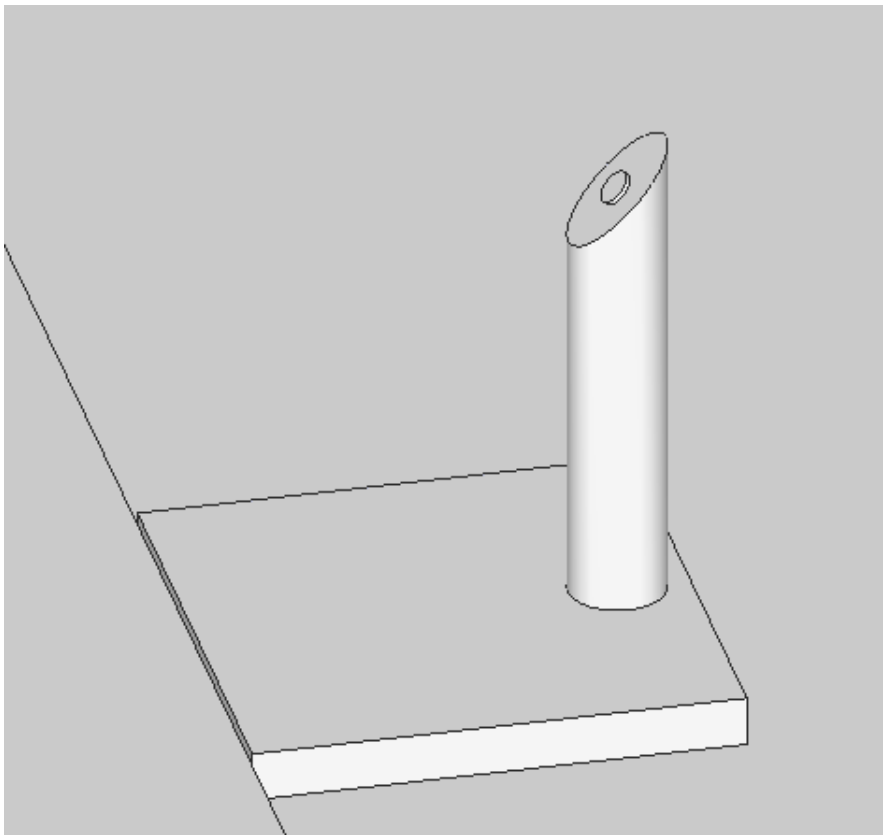
them with a slope of 30°. And finally make a platform (4) that will be an elevator between the bloc (2) and the goal.



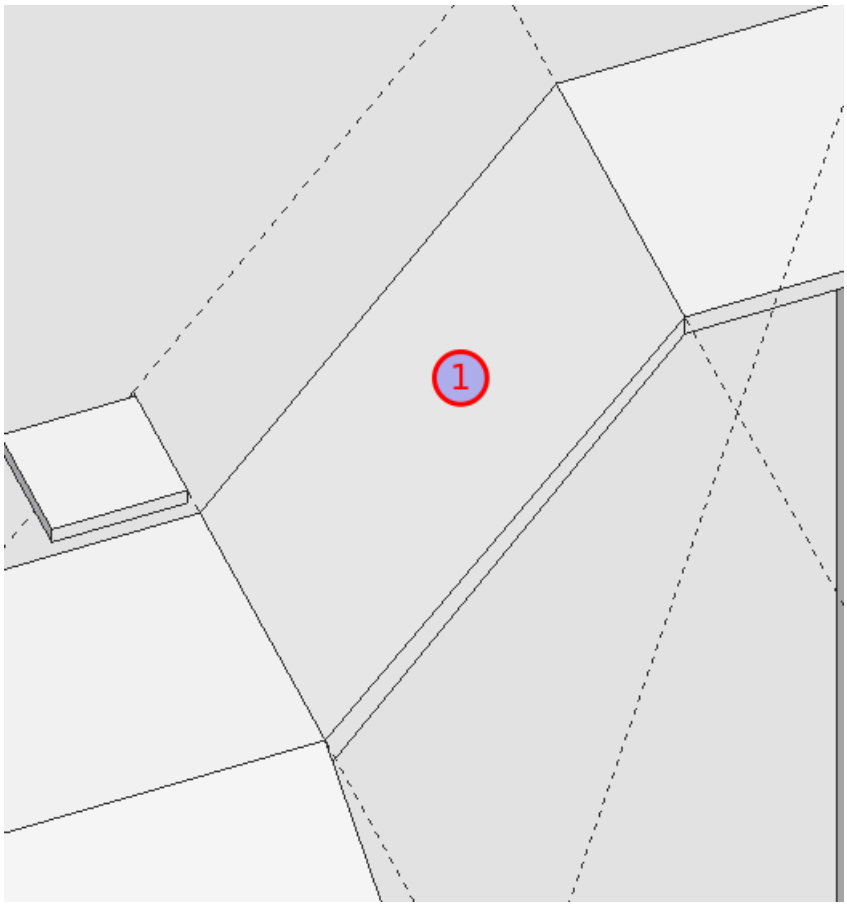
10) Draw another platform that will travel between (1) and (2).



11) On this platform draw a button that will activate the platform/elevator.



12) Make a bridge(1) between the platform of the goal and the second bloc. This bridge must have a steep slope ($>45°$).



13) Be sure to remove all construction lines.

Hint

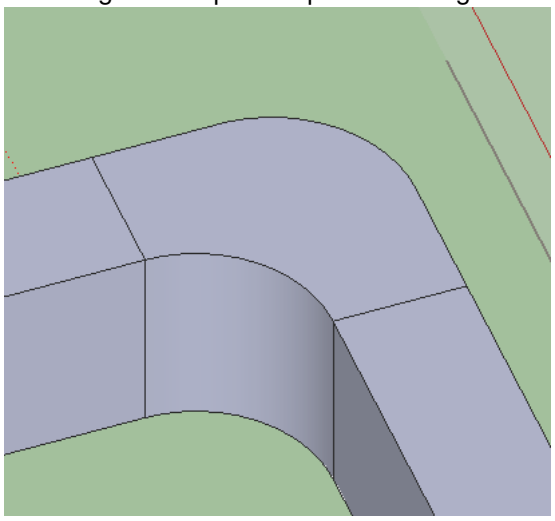
No need to add textures here, since the export doesn't include them.

Hint

Sketchup gives by default a little character and a unit for dimensions. You can use them to estimate appropriate dimensions for your building. Personally I use the meter unit. It is possibly anyway to resize it later.

Hint

When you draw round shapes, expect Sketchup to make lot of faces, since the whole mesh will be converted to triangles. It will be easier to work with the mesh if those areas are small. With the pencil tool, you can draw edges to help the exporter making cleaner faces.

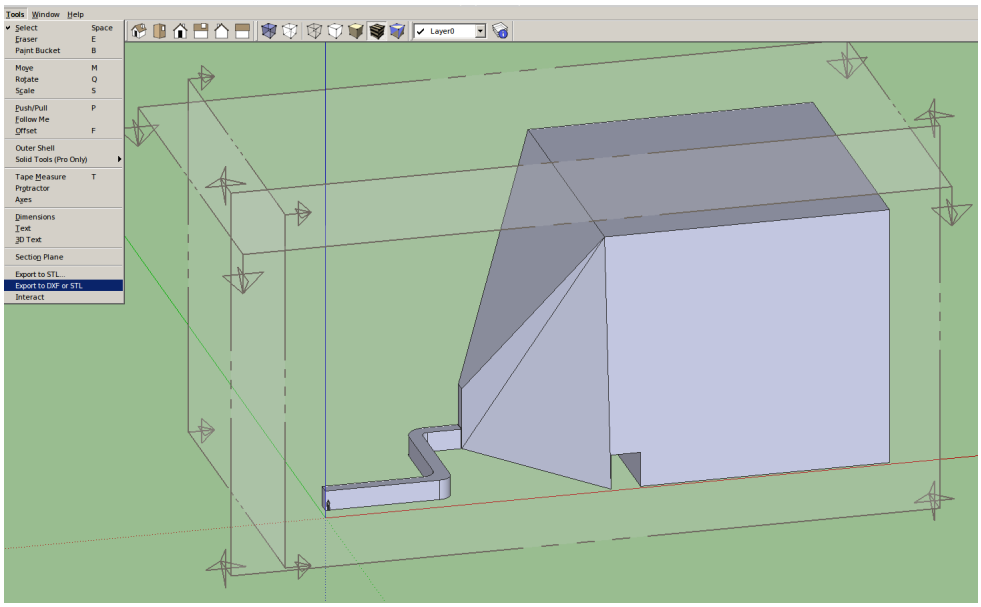


3.2 Export to STL

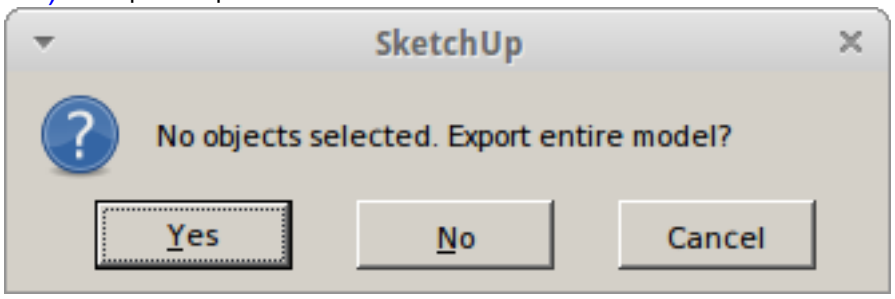
Important!

Delete or disable all section planes, since only what's visible is exported.

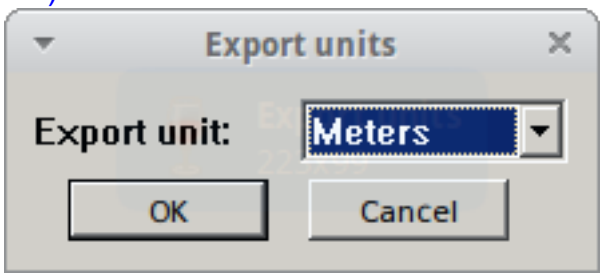
14) Be sure nothing is selected and go in menu Tools / Export to DXF or STL



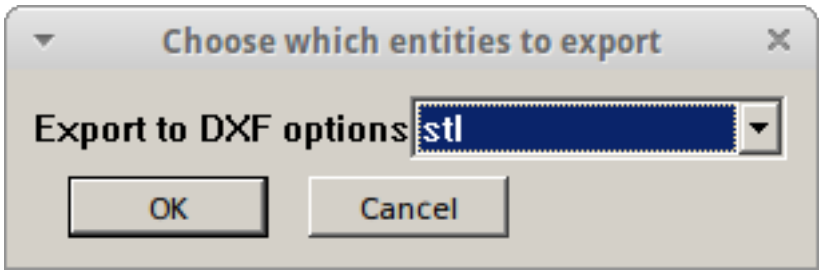
15) Accept to export entire model



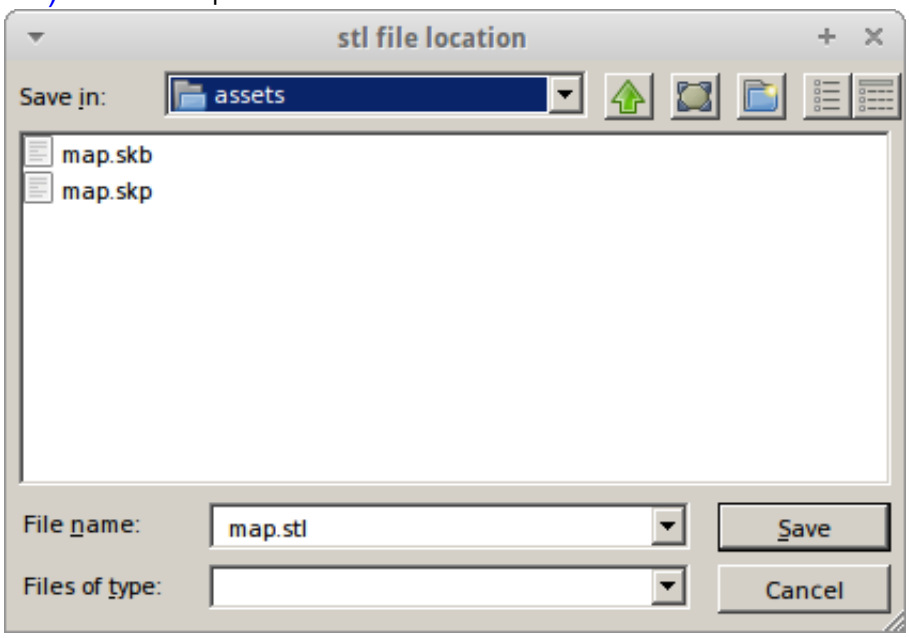
16) Choose Meter unit



17) Choose stl



18) Save to ”map.stl” in the ”assets” folder



Now the map is exported to STL format and can be imported in Blender.

4 Prepare the map in Blender

4.1 Import STL

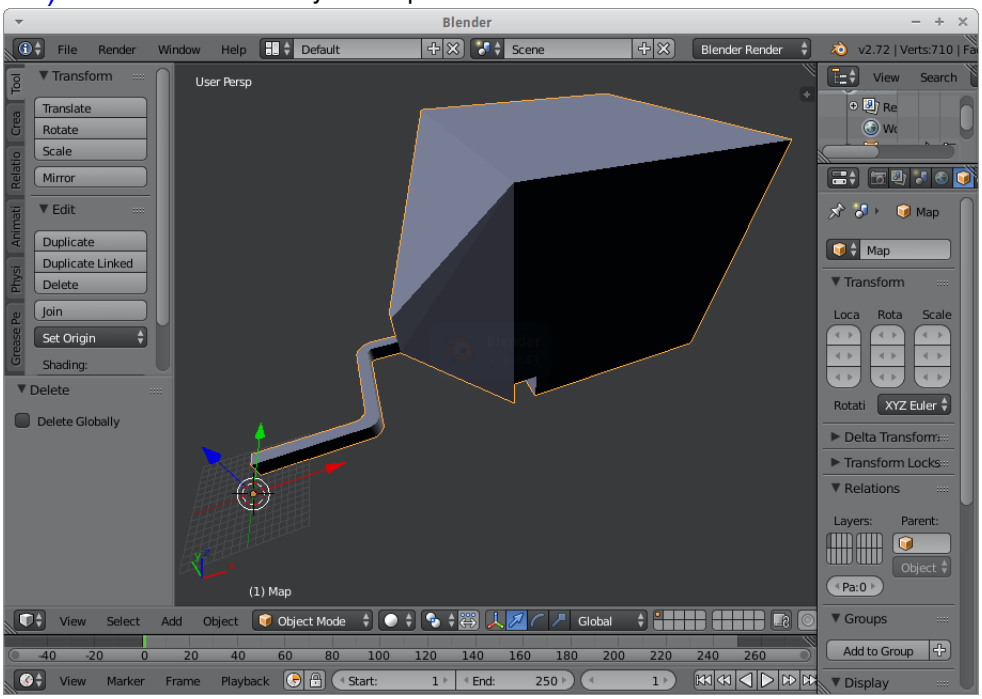
19) Start Blender

20) Remove everything **[I,I,X]**

21) Import the STL file (menu File / Import / Stl). Use the default options.

Hint
If the STL importer is not in the list, check that the STL format plugin is enabled in the User Preferences / Addons.

22) You should now see your map.



23) Save the file in the same folder. **[Ctrl-S]**

4.2 Normals

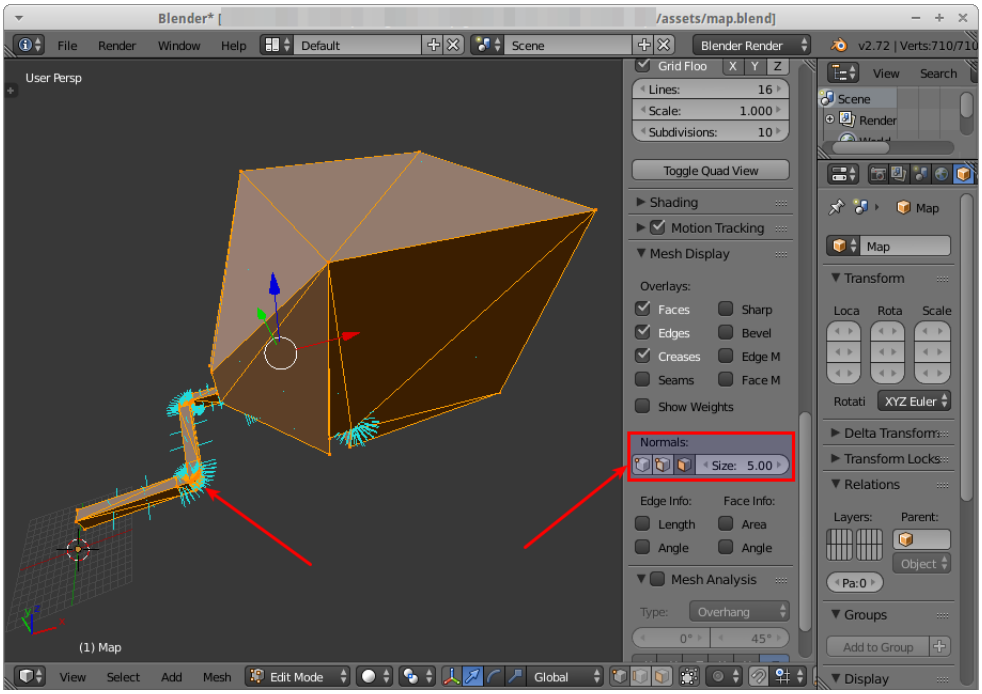
We must now check that all face normals are properly oriented inward.

24) Select the mesh

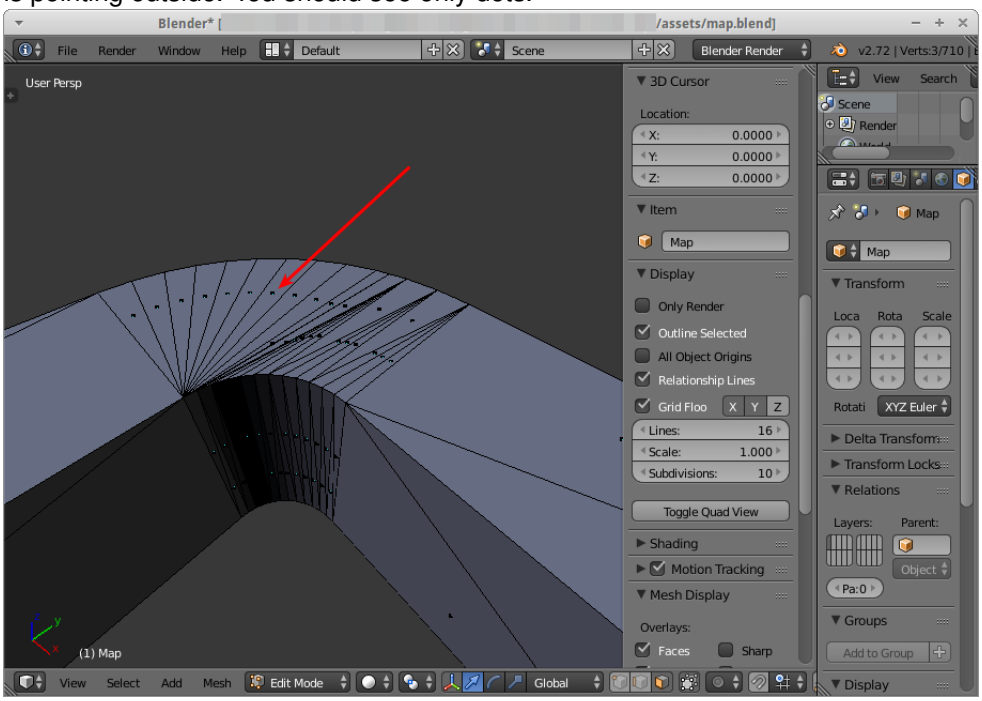
25) Enter in edit mode **[Tab]**

26) Open the properties panel **[N]**

27) Look for Mesh Display and activate the Display face normals as line. Set the size to something visible, like 5. The normals should show up as cyan lines.



28) Reduce the size of the normal, like back to 0.1, and check that no normal is pointing outside. You should see only dots.



Hint

If you find a face with the normal pointing outside, you can invert it with menu Mesh / Faces / Flip Normals. **[Ctrl-F,F]**.

29) At the same time, you can remove unwanted faces or vertex.

Hint

You can hide faces with **[H]**, and show all hidden faces with **[alt-H]**.

30) Disable the display normal when you're done.

4.3 Separate elevators

The elevators made in Sketchup are actually part of the map. But in Godot they need to be separate meshes.

31) In edit mode, select all vertices of one elevator and separate it with **[P,1]**.

Hint

You can easily select all vertices by selecting one and select all linked vertices **[Ctrl-L]**.

32) Repeat the previous action for each elevator.

4.4 Textures & UV mapping

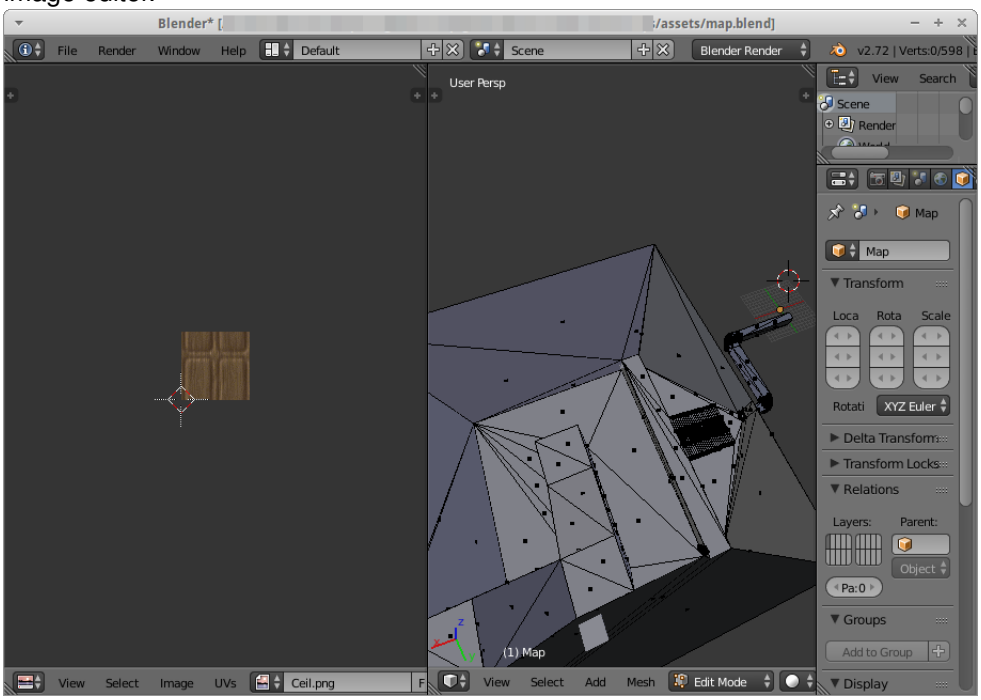
33) Go back to object mode **[Tab]** and open the UV/image editor.

34) Open each texture from the "assets" folder to load them in memory.

35) Go back to the 3D View, select the map mesh and go in edit mode **[Tab]**.

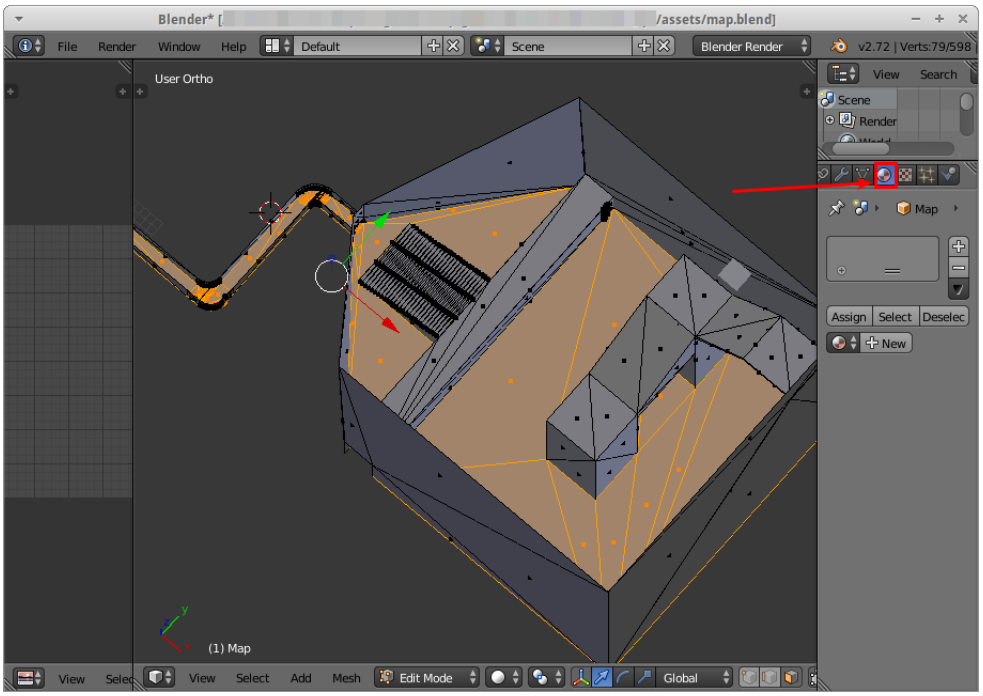
Hint

For convenience, split the view in 2 and keep visible both 3D View and image editor.



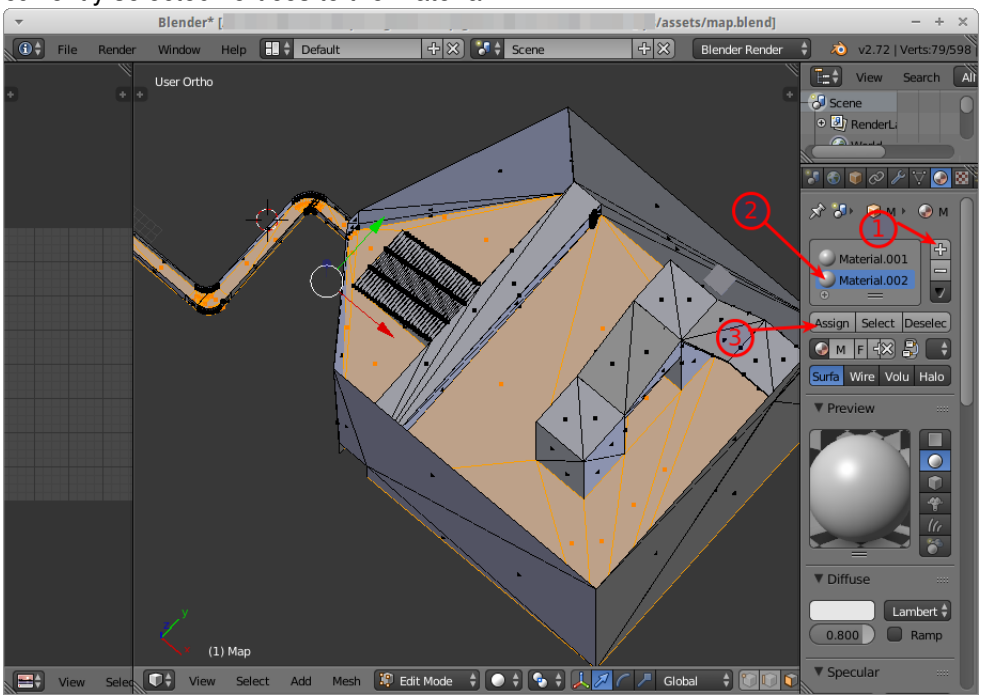
36) Select all faces that are the same kind, for instance the floor. They will have the same texture.

37) In the Properties view, select the Material Tab.



38) Create 2 material slots (1) and a new material for each.

39) Select the second material slot (2) and click ”Assign” (3) to assign the currently selected vertices to the material.



40) Rename the first material slot as ”Wall” and the second material slot as ”Floor1”.

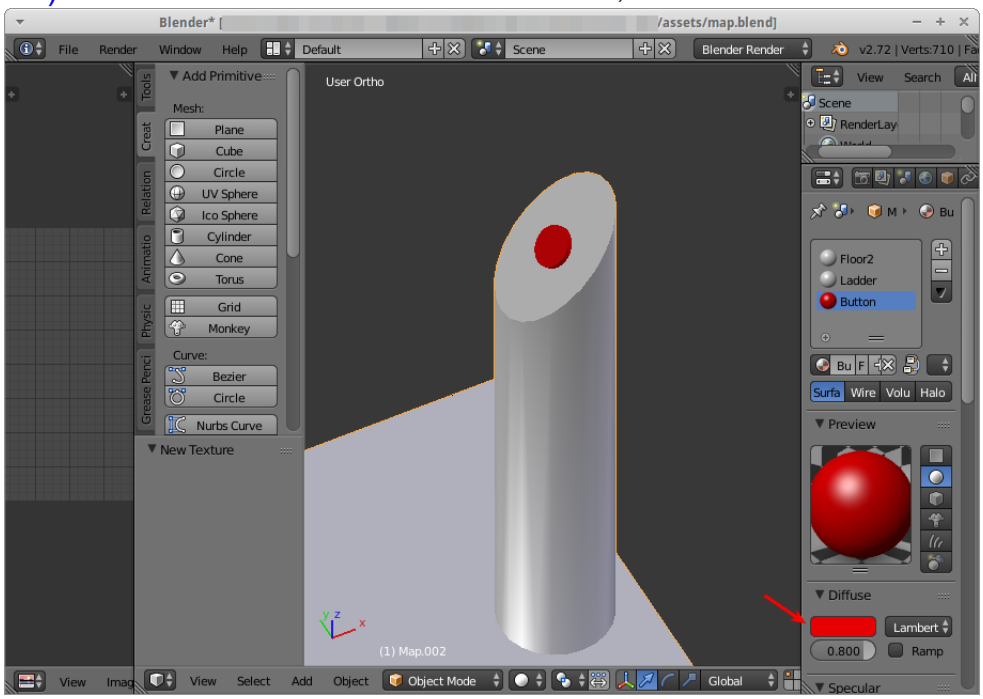
Hint

The first material slot created is by default assigned to the whole mesh. That’s why I selected the second material slot to assign the vertices.

41) Create as many new material slots as there are textures, except for the two already created. And assign the vertices that should be linked with the texture. In my case, I have 1 wall texture, 1 ceiling texture and 2 floor textures. I had to create 2 others slots for the ceiling and the other floor textures. On top of that I created a special material slot for the ladder, without texture but with a gray color.

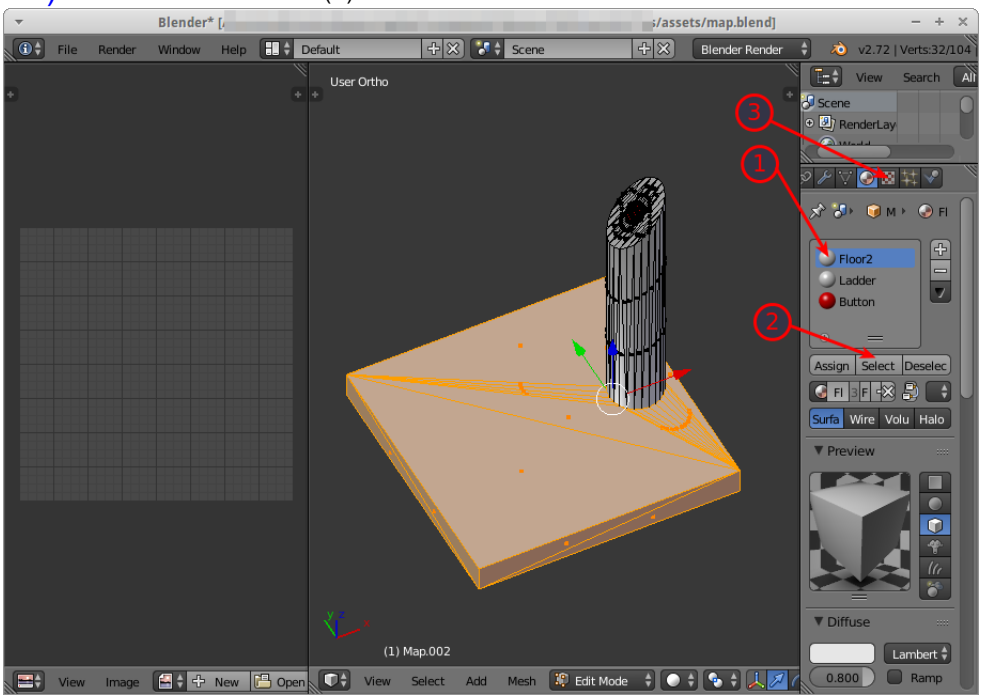
42) Repeat the same actions for the elevators. You can however reuse existing materials. I had still to create a new material for the button, which I wanted to be red.

43) For the button material and the ladder material, set some colors.

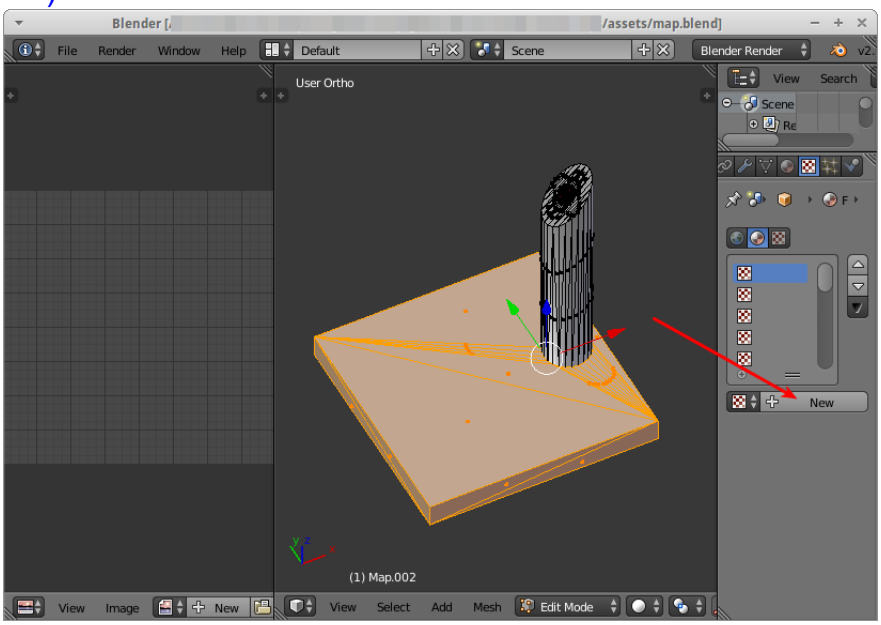


44) Now, we’ll apply some texture. Select the material slot that will have a texture (1) and click ”select” (2) to select all associates vertices.

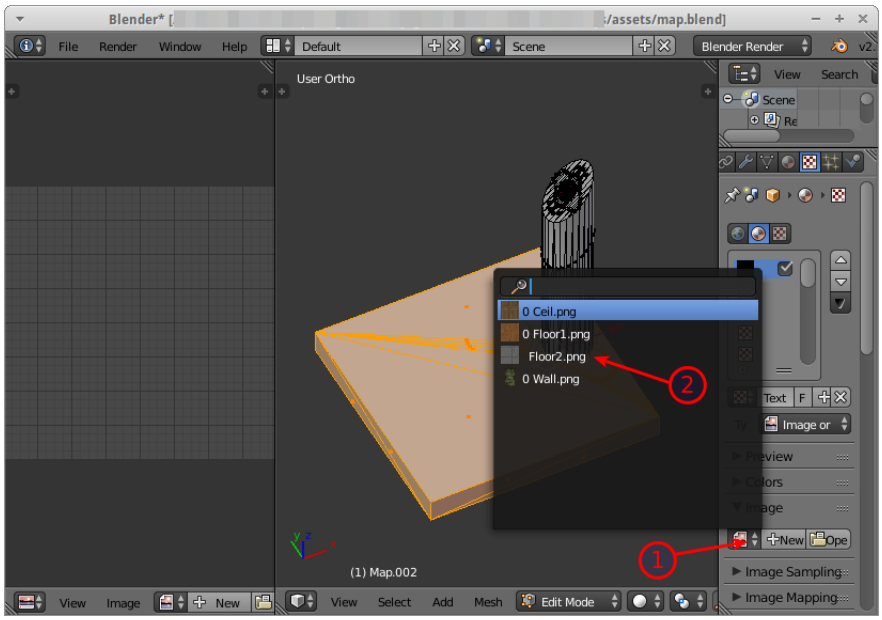
45) Go in the Texture tab (3)



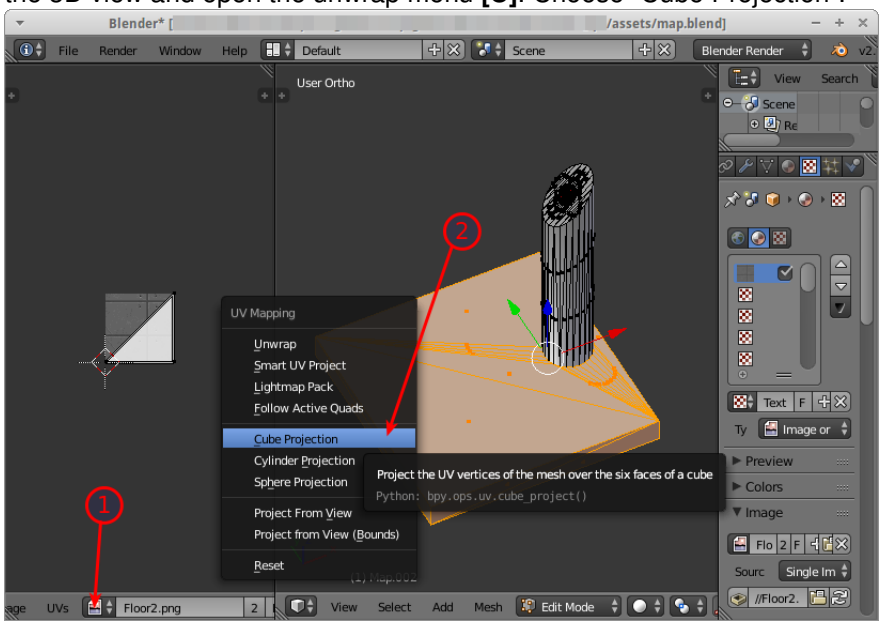
46) Create a new texture



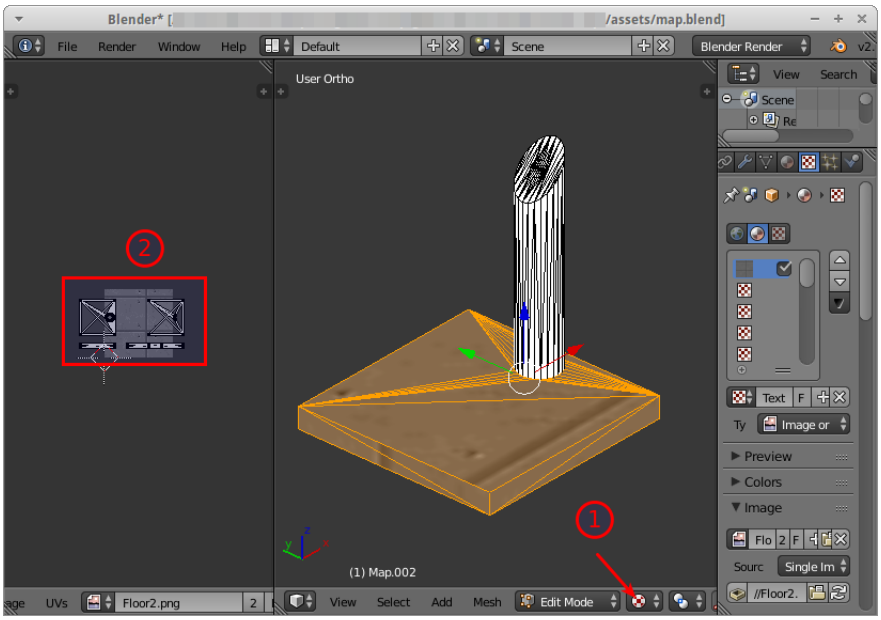
47) Open the list of loaded images (1) and choose the appropriate image (2)



48) In the image editor, select the same image (1). With the mouse, go in the 3D view and open the unwrap menu **[U]**. Choose ”Cube Projection”.



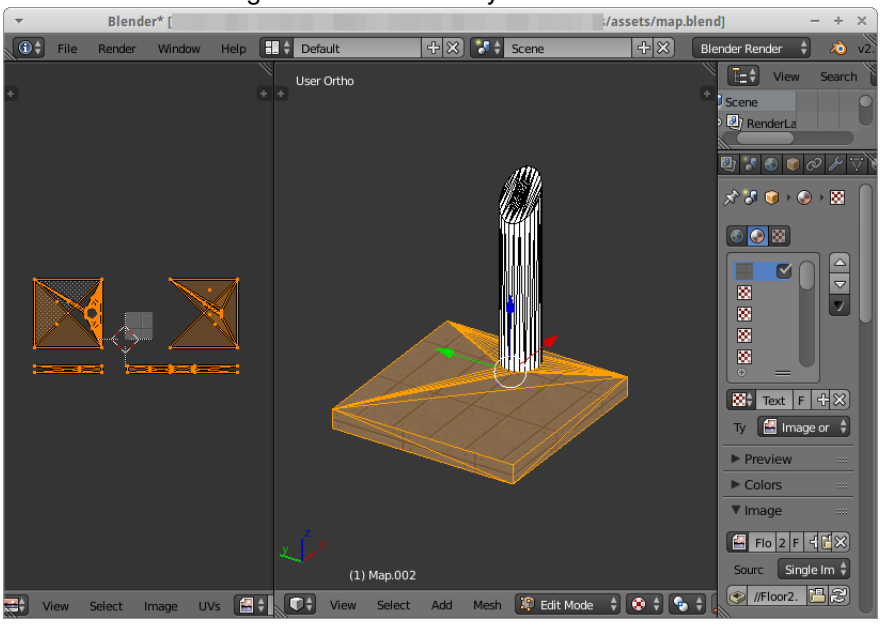
49) The mesh is now UV-mapped. You can see the result by selecting Texture in the Viewport Shading menu (1). In the image editor (2) are showing up the UV mapping of the select faces.



Hint

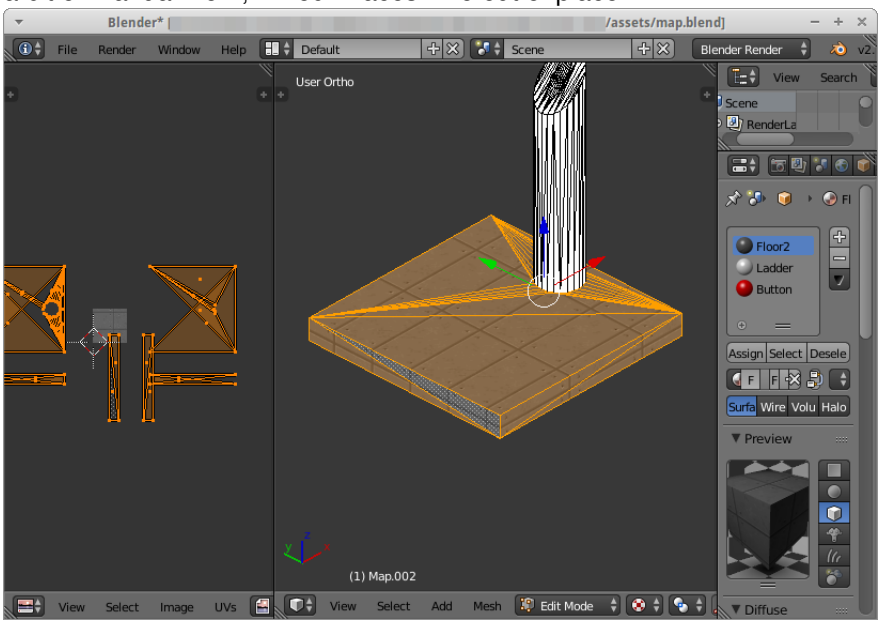
The Cube projection is especially useful here because it calculates an UV mapping appropriate for walls and make a proper synchronisation between walls so it looks continuous.

50) The UV map is too small. We want the texture to be smaller and to repeat itself. In the image editor, select all faces **[A]** and scale **[S]** until the texture is small enough. I scaled x5 for my elevator.



Hint

Sometimes the UV unwrap doesn't put the faces together, which results with a weird mapping when scaling the faces. But if you scale with a round number (by typing the number), it should always look fine. In the worst case, you can always manually correct the position of the faces, or use the "Project from View" unwrap option for the faces you want. With a bit of manual work, I fixed 2 faces who out of place.

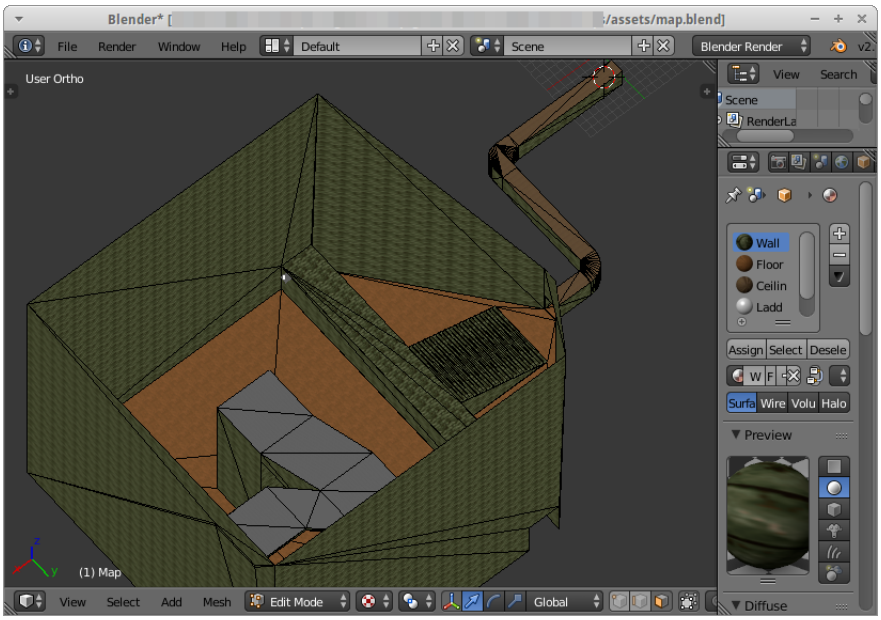


Hint

The button and the tube are white even though they have a material. Don't panic, it's normal. The Texture View Shading shows only textures which are UV mapped, and materials with only a color are rendered in white. In the solid mode and the final rendering (if you add some lights), it looks fine.

51) This mesh is now ready. Repeat the same process for all others meshes. You don't need to recreate a texture for the material slot you already assigned to a texture, because the material is shared between the meshes.

52) After that, your map should look a bit like this:



Once all meshes are UV mapped and all materials have a texture, this section is done and only remains the lights and the door.

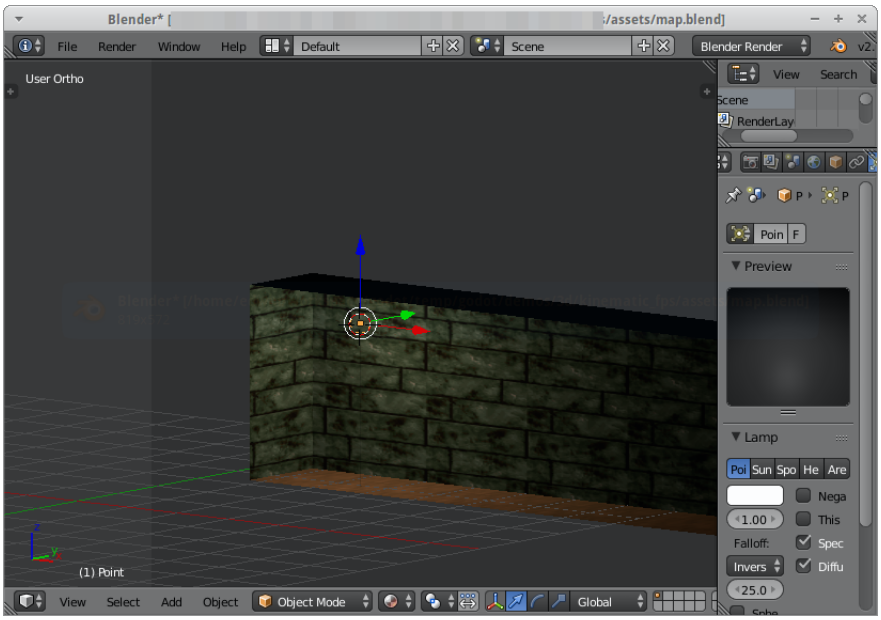
4.5 Lights

Adding light is very simple, but at the same time it can be a whole profession on itself just to get the proper lighting. But here we'll only see the simple version.

53) Set the 3d cursor to where you want to add a light.

54) Add a point light with **[Shift-A]** and select Lamp / Point.

55) Change the Viewport Shading to Material so you can see the result.



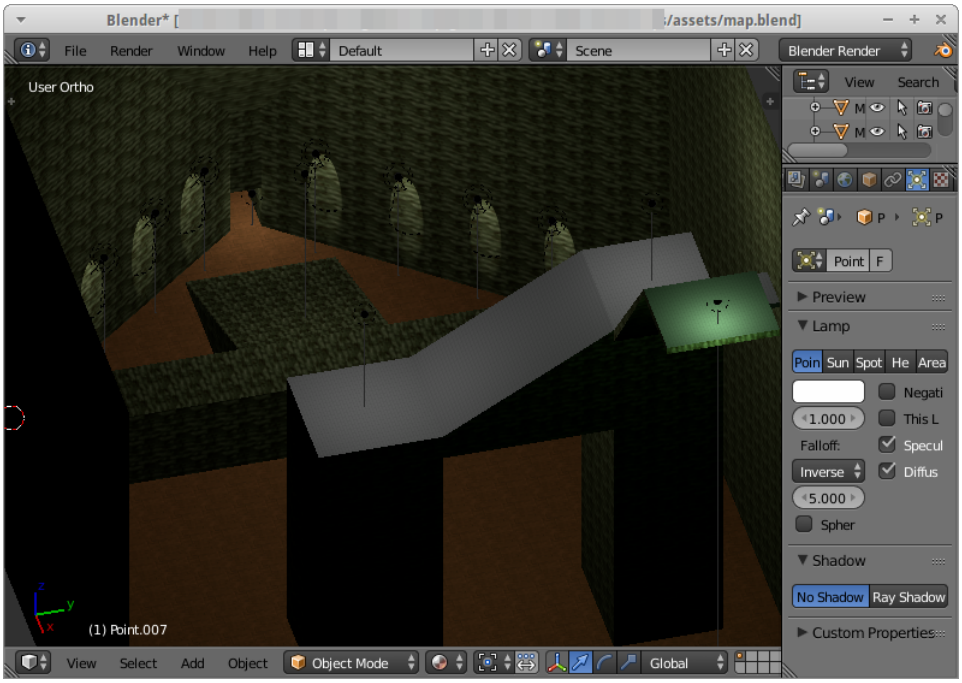
56) Change the Energy of the lamp and the color to your content.

Hint

If walls and floors seem to shine like plastic, reduce the specular intensity of the material.

57) Repeat as much as needed.

You have now a map with lights. Something like this:

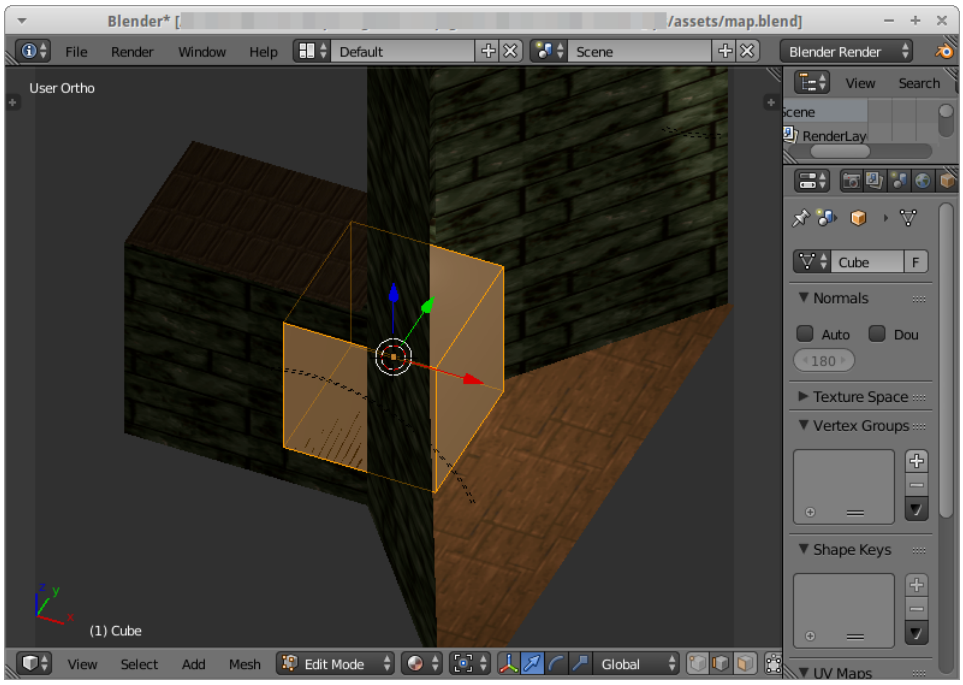


Now we're almost done with the assets.

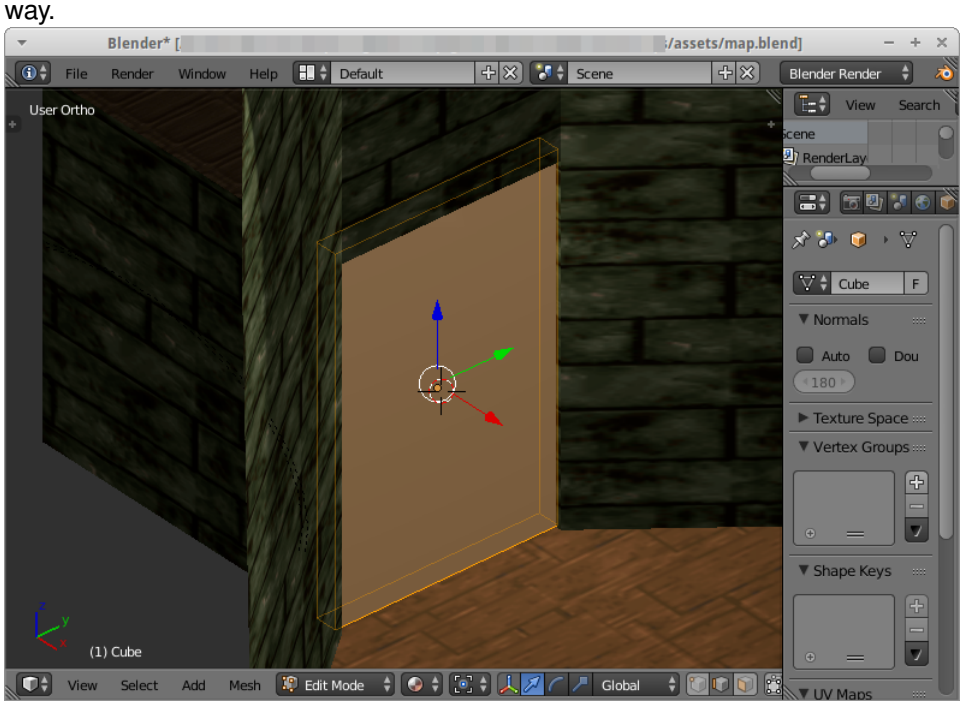
4.6 Doors

Since we're at it, let's make a door. Just a simple one, like in the old game Doom[9], which goes up when we activate it.

58) Put the 3d cursor to where you want the door and add a cube.



59) Go in edit mode and resize it to a vertical rectangle that covers the hallway.



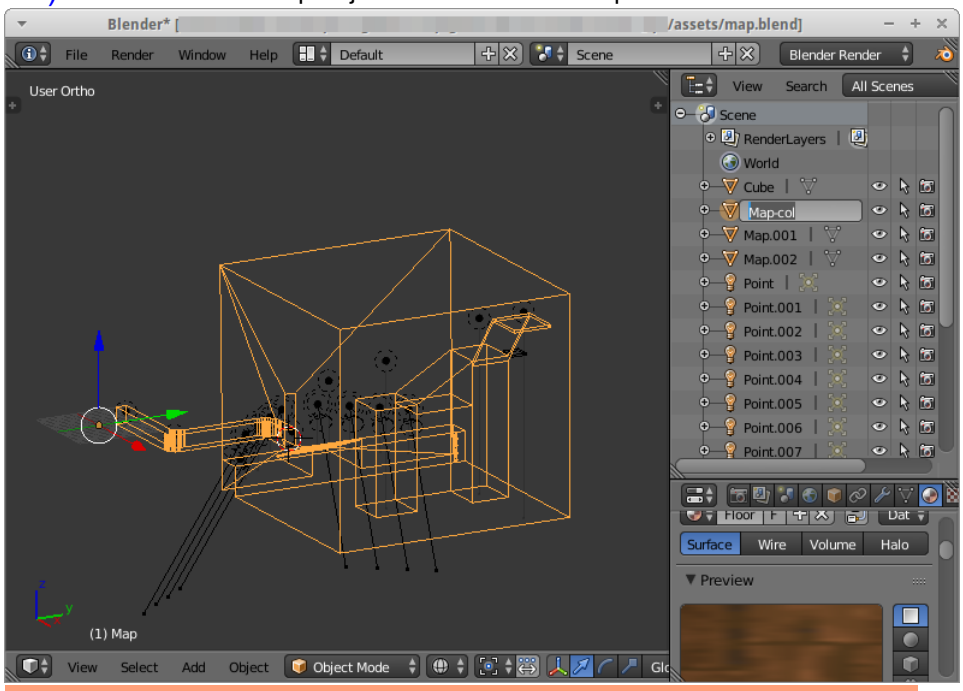
60) Give it a material and a color (or a texture, if you want).

That's all.

4.7 Names

Before exporting the map to Godot, we must give a proper name to each mesh we created. Lights don't need a special name.

61) Select the main map object and rename it "Map-col".



Important!

You must add the "-col" at the end of the name. The importer of Godot will understand that it is collidable, and it will automatically create a rigidbody node for the mesh.
You can check the wiki of Godot to know more about the others available options.

62) Select the elevator meshes and rename them "Elevator1-col" and "Elevator2-col".

63) And finally rename the door as "Door-col".

Hint

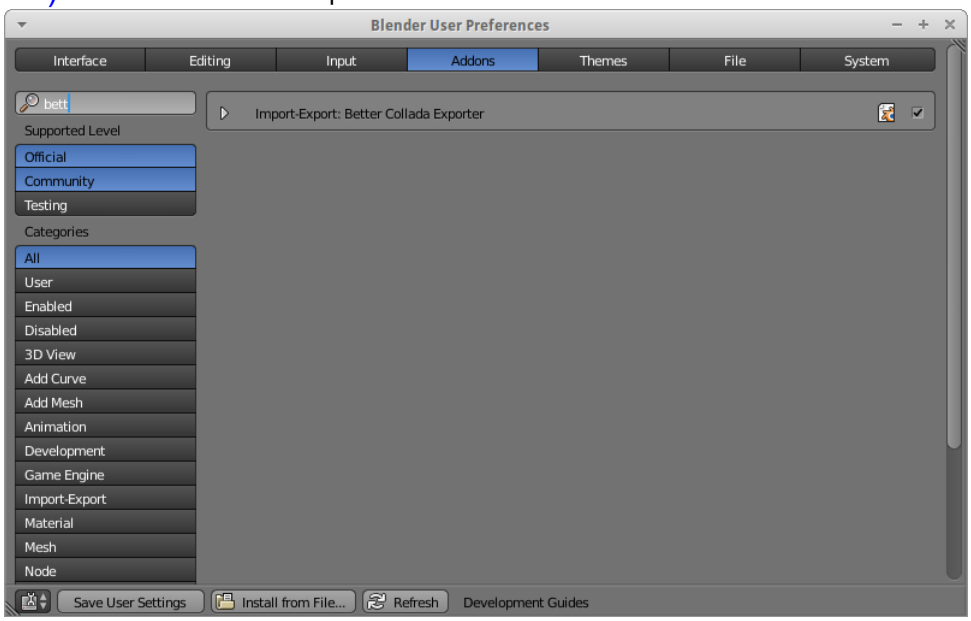
You are free to choose whatever name you want. It is only important to keep the options like ”-col” at the end as a suffix. Those suffixes won’t appear in the name of the node in Godot. You can also combine options by adding them like this ”My_object-col-option1-option2-option3” (the name in Godot will be *My_object*).

4.8 Export to Collada

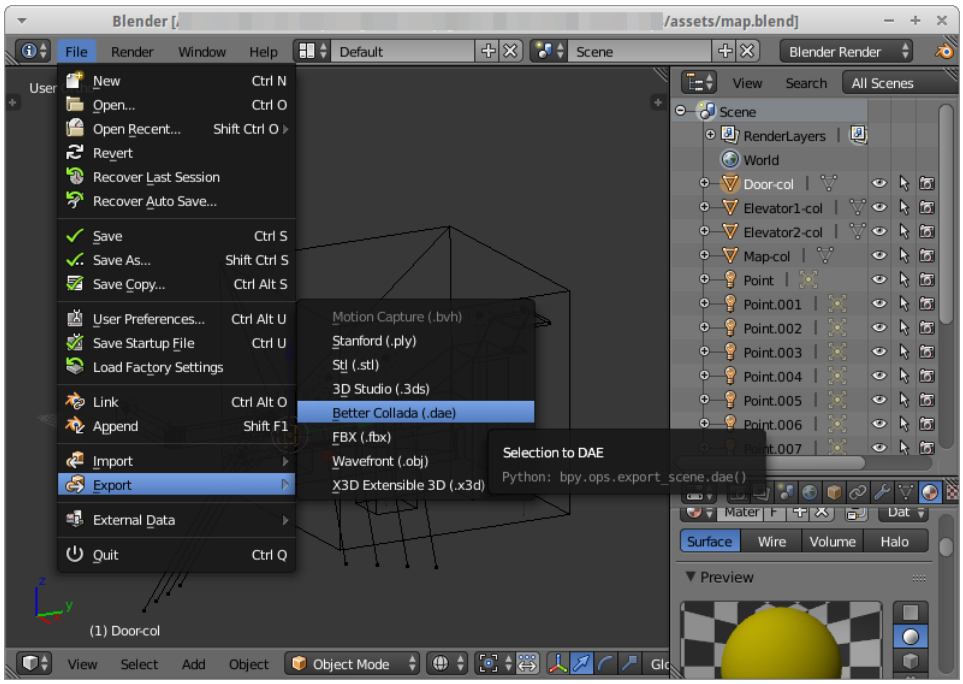
Now the map is finally ready to be exported.

64) Save! If you didn’t save your file until now, it is the time to do it **[Ctrl-S]**.

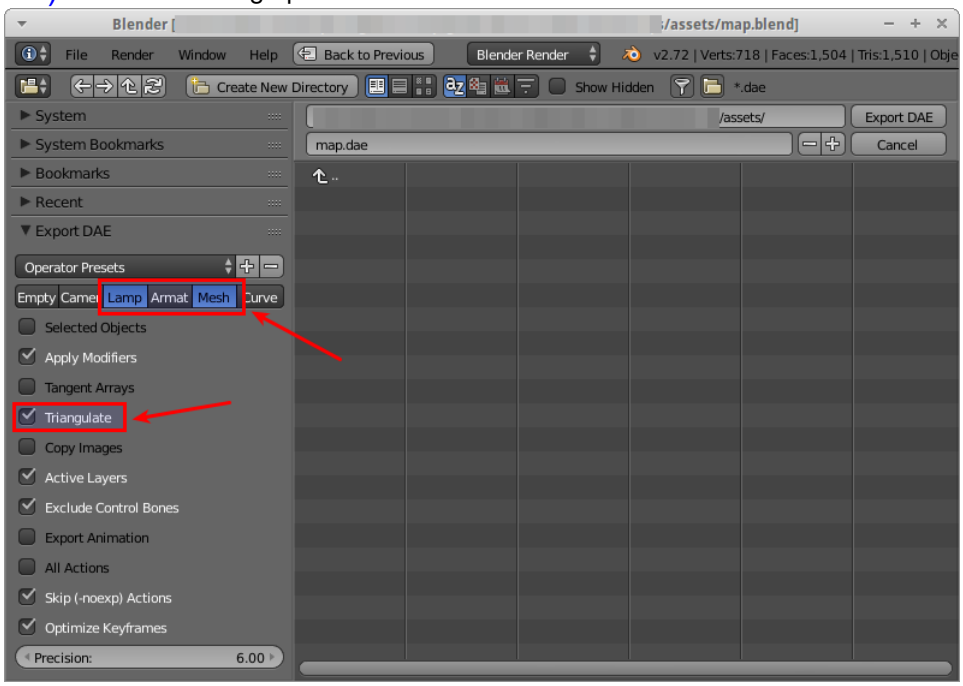
65) Enable the Collada exporter distributed with Godot if it’s still not enabled.



66) Open the menu File / Export / Better Collada (.dae)



67) Set the following options:



68) And then click the ”Export DAE” button.

69) You can now close Blender.

70) You should have a .dae file in the ”assets” folder. Copy this file in the ”game_assets” folder.

That’s it! Now you have a map, with elevators and a door. There’s a button too, but it will be defined in Godot. Next, we’ll import the scene in Godot and start with the main dish : physics!

Hint

You can find my .dae file and the texture in the ”source/game_assets” folder of my project in Github. And you can find the blender file in the ”assets” folder, if you want.

5 Import the map in Godot

71) Open the project in Godot.

72) In menu, Import / 3D scene.

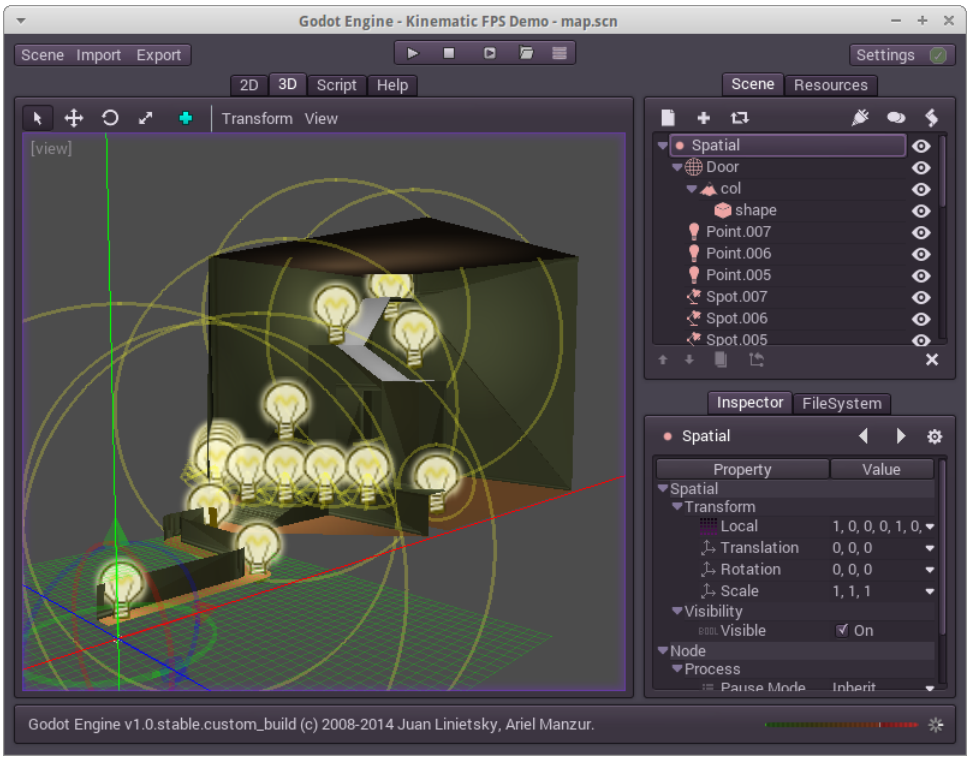
73) Select as source scene the .dae file in the ”game_assets” folder.

74) Select ”res://game_assets” as target path

75) Disable Import Animations

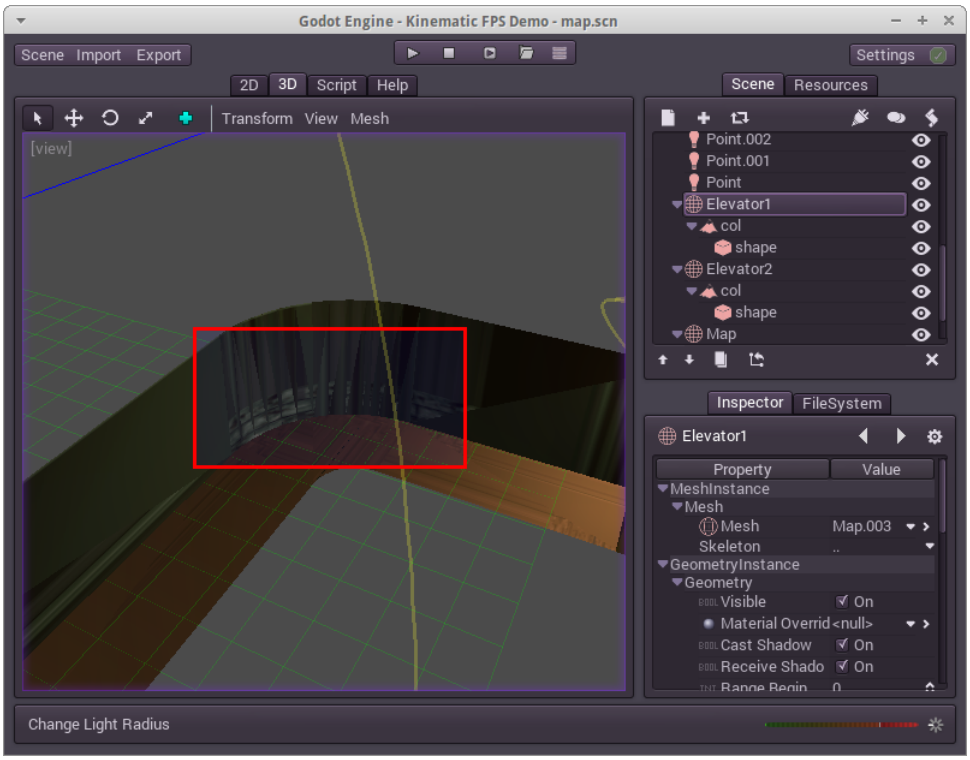
76) Click on ”Import & Open”

77) You should now see the map



78) Disable the default light in menu View, and adjust the lightning.

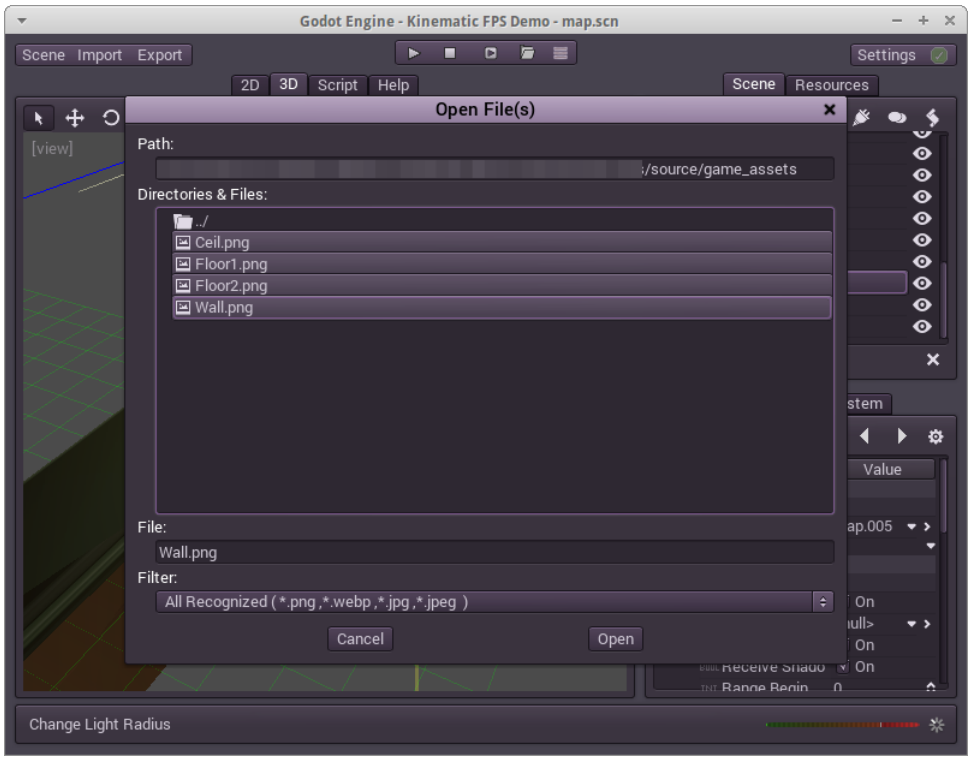
You have maybe noticed that the textures seem off. At best only a part of the texture shows up.



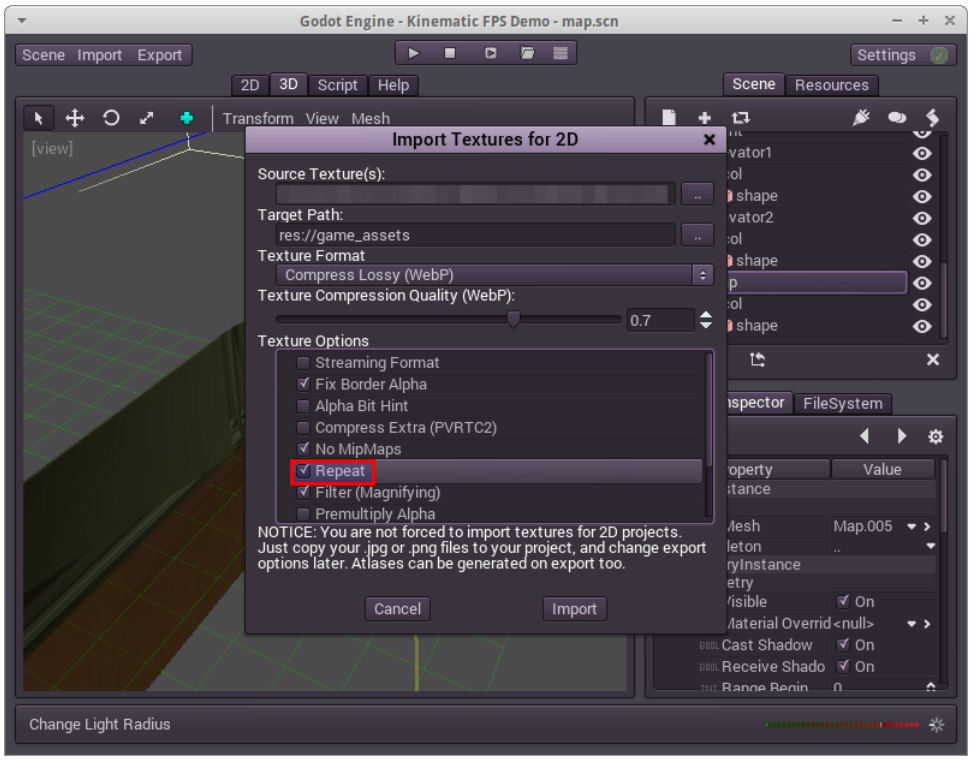
That's because the texture is not repeated. We'll fix that.

79) Go in menu Import / 2D Texture.

80) As source, select all textures at once in the "game_assets" folder.

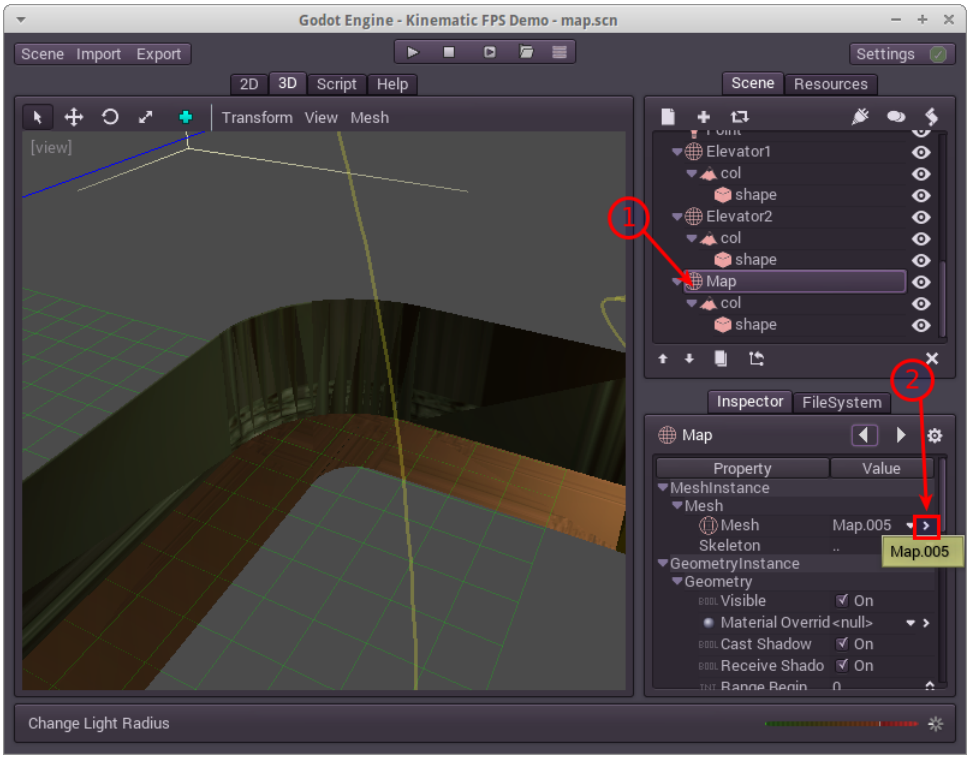


81) Select ”res://game_assets” as target and check the repeat option.

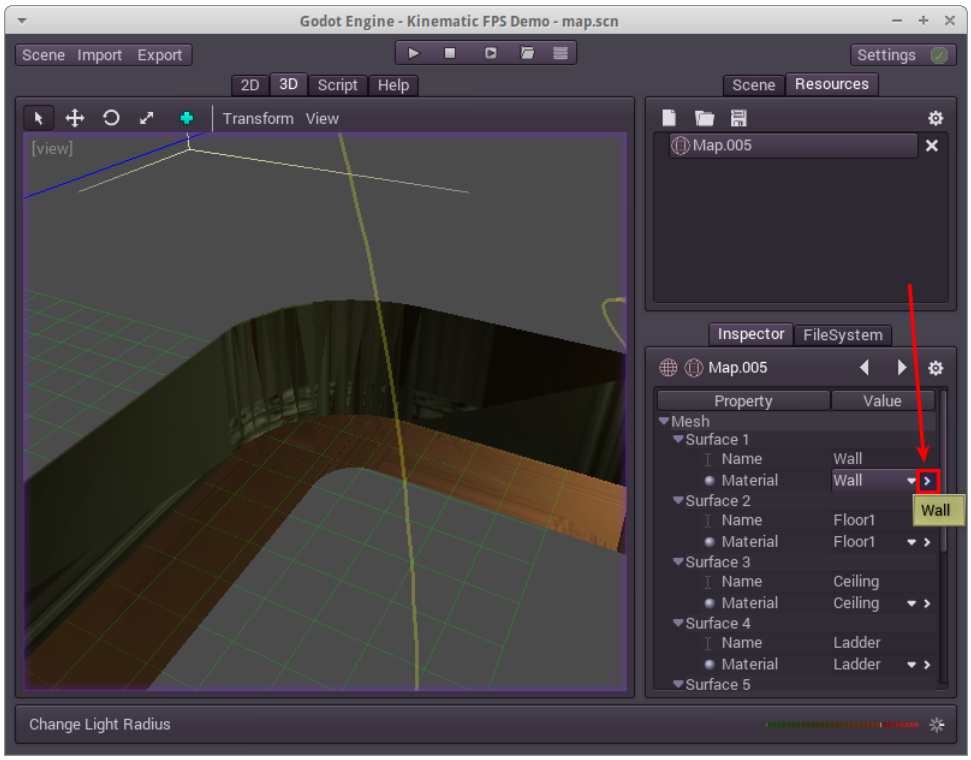


82) Import.

83) Select the node of the map (1) and open its mesh property (2).



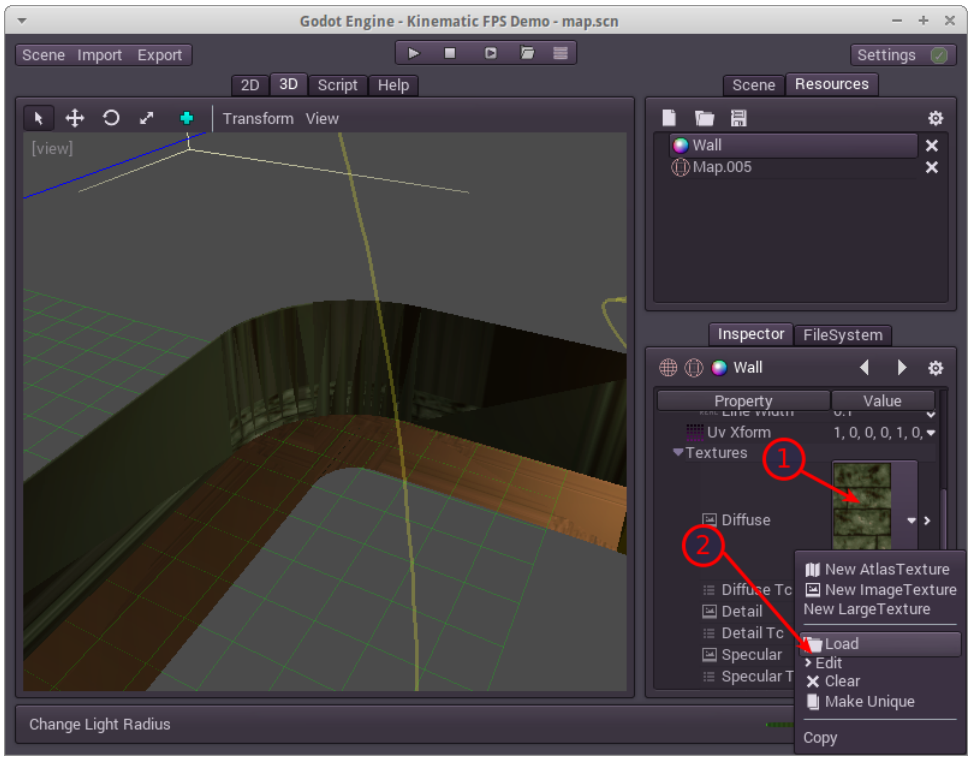
84) You'll find multiple surfaces, which are the equivalent of materials in blender. Select the material of the first surface with a texture and open its details.



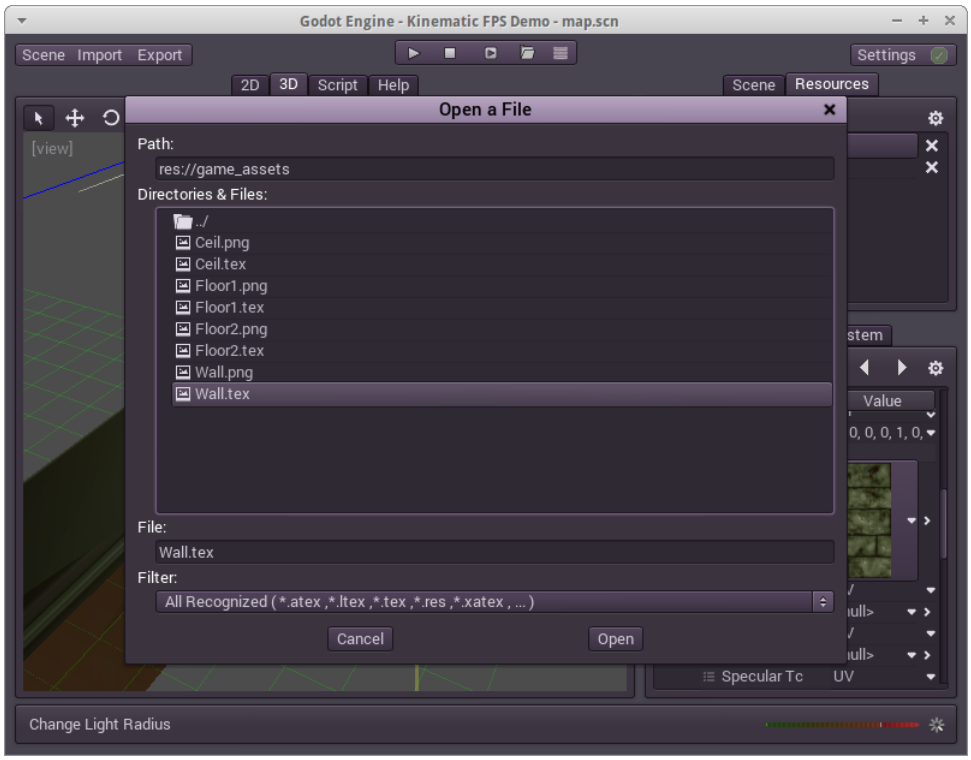
Hint

During the import, the name of the material you gave in blender shows up here.

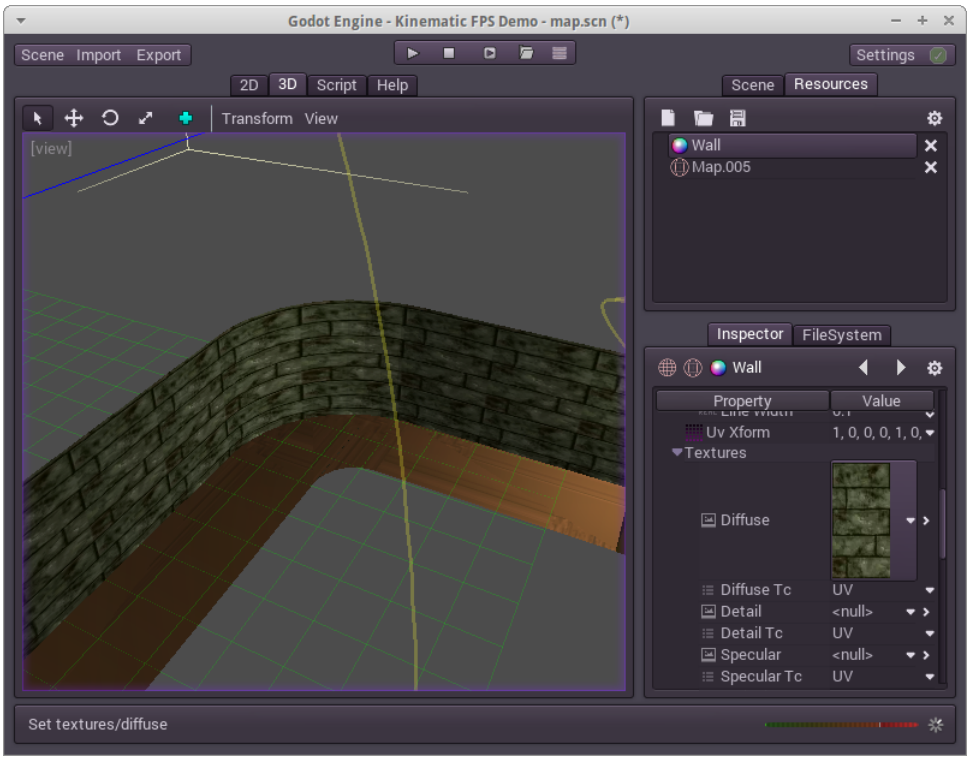
85) You can see here all the options of the material. Go to the option Textures / Diffuse (1) and click on ”Load” (2).



86) Then choose the appropriate file with a .tex extension.



87) The texture is now properly repeated.

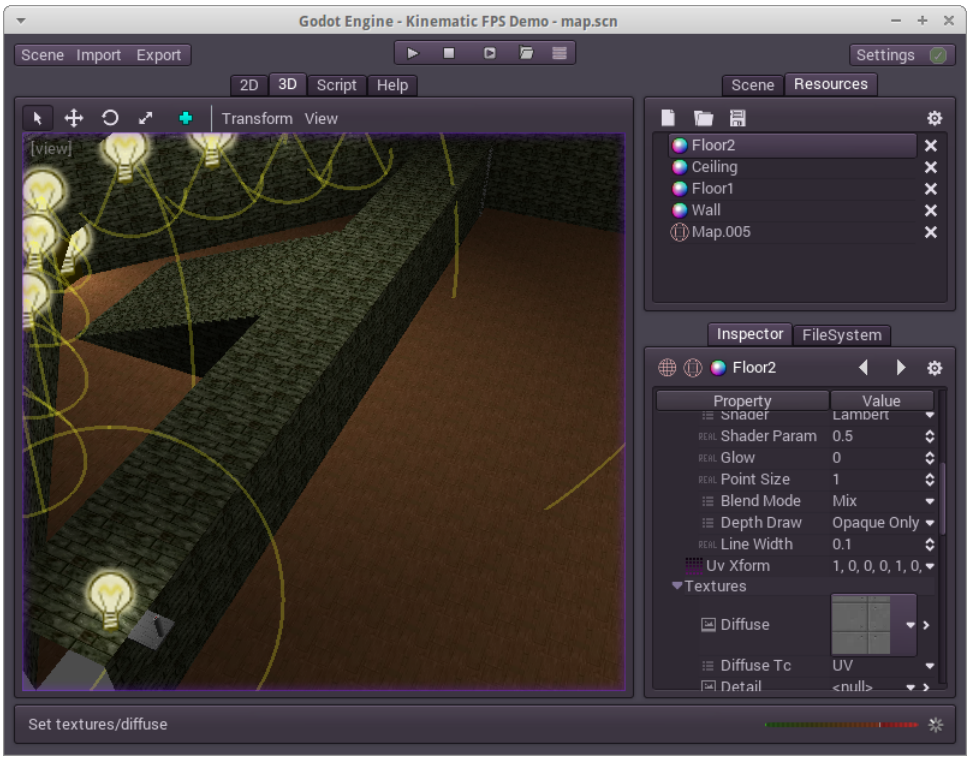


88) Repeat the steps for each material of the scene.

Hint

No need to do it for materials without texture.

At the end, you have a properly textured map.



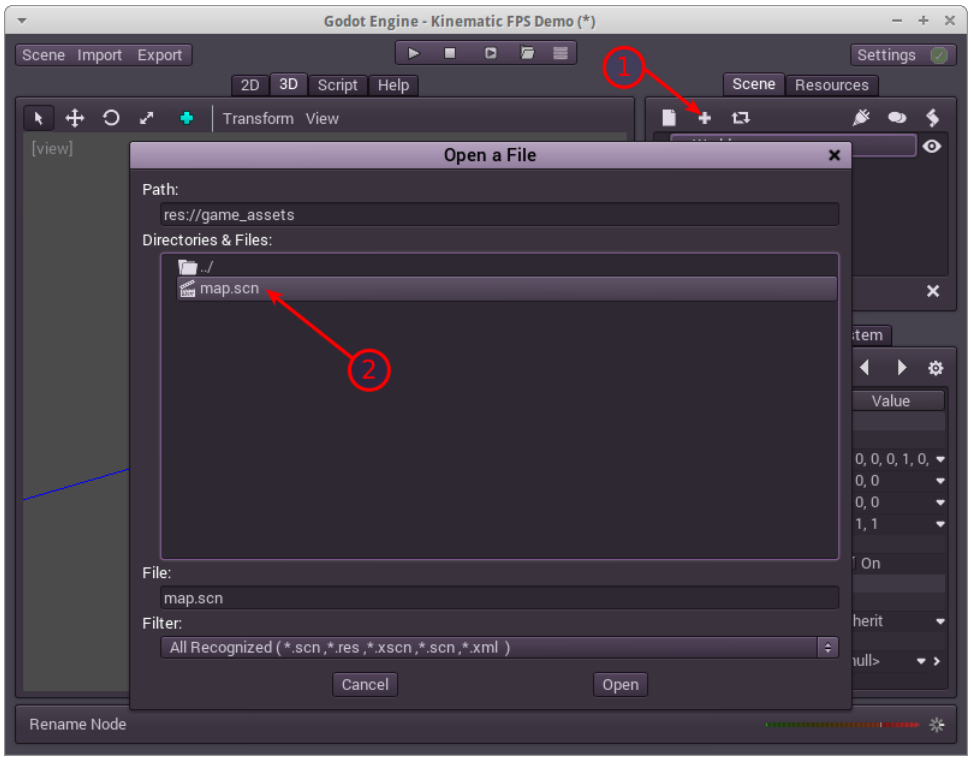
6 Main Scene

Now that we have imported everything, let's create the main scene. It's better to modify as little as possible the imported scene because everything would be lost if you have to import it again.

89) Create a new scene in the menu Scene / New Scene.

90) Add a *Spatial* node and name it *World*,

91) With the *World* node selected, add an instance of a scene (1) and select the imported scene (2).



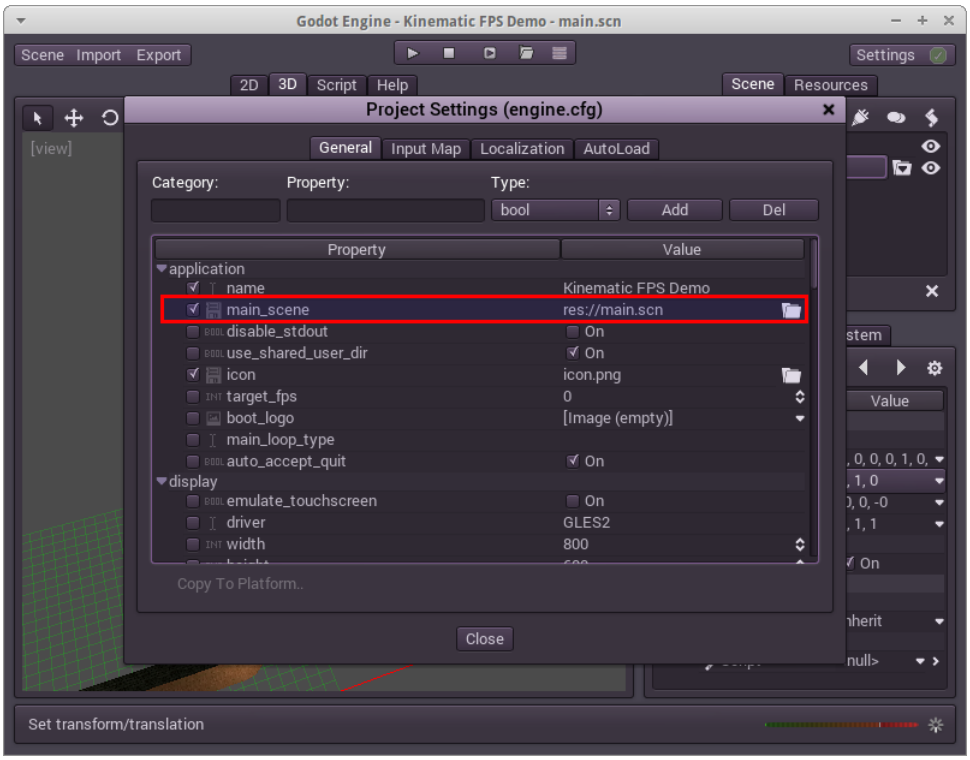
92) The map is now showing up. Move it a bit vertically, for instance to y=1.

Hint

It's a small fix to a strange bug that happens when the player is at position y=0. It makes the player vibrate endlessly. Later, if you see such problem, elevate the map even more.

93) Save the scene to "res://main.scn".

94) While we're at it, go in the project Settings and set the main_scene to "res://main.scn".



Now we have a map and a main scene. We can finally work on the player.

7 Player

7.1 Theory of kinematic objects

Before we start, a bit of explanation.

Physics engine Godot contains a physics engine that provides tools to detect collisions and calculate paths to slide along the colliding objects. It doesn't provide however a ready to use solution for a FPS character that can walk, jump and climb stairs. And that's what we're going to implement in GDScript.

Rigid body and kinematic body In Godot, a rigid body is an object that is managed completely by the physics engine. We can only manipulate it by giving force impulses to move it or set its position. But eventually it's moved by the physics engine and the result is hard to predict. The kinematic body is like a rigid body except that we have to make it move ourselves with a script. We must take care of its velocity, collision detection, gravity and everything. However we can use functions from the physics engine. And the most important one is *move(...)*, which makes the body move to a specific direction, but only if it doesn't collide with something else. With this, we are certain that our object

would never intersect with another object. It requires more work to use a kinematic body rather than a rigid body because we must take care of everything, but we have a complete control of our object.

Skeleton of a kinematic body Technically a body can have any shape. In our tutorial, our character is a human. It doesn't need to be complex and a mere basic geometric shape is enough. In fact a mere box would be enough in most cases. If we take for instance a 2D platform game like Mario Bros (see fig. 1), the character collides with the ground and enemies as if it was just a rectangle (in yellow). You can also notice that in such game the rectangle never rotates or get transformed. Only its position changes. In similar 3D games, it's the same concept, except that we use a box instead of a 2D rectangle.

Collision and sliding The physics engine calculates automatically the trajectory of the box by giving it a velocity. And if it encounters a wall, it will try to slide along the wall. At least, as much as possible. Because it the character tries to run directly against a wall, he won't slide at all. But if the velocity of the character goes a bit along the wall, the calculated velocity to slide will be perpendicular to the wall. As shown in figure 2, the object at position P moves with an initial velocity V and collides at position P_1. A sliding velocity S will be calculated and the object will move to its final position P_2.

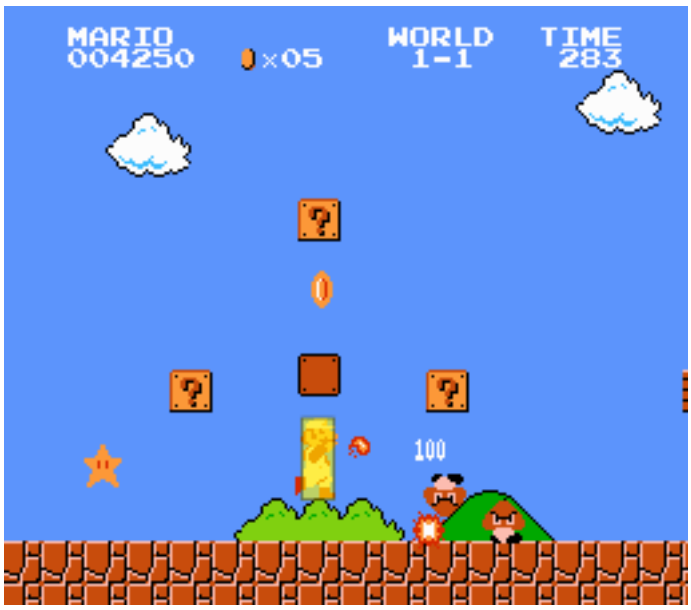


Figure 1: Mario Bros on NES

Problematic of the staircase With the sliding, a mere box is enough in most cases. Even with a slope, the box would be able to properly slide against it. However there is one case were the box doesn't suffice. Stairs. We humans can climb stairs thanks to our legs, but a box that slides against the floor would

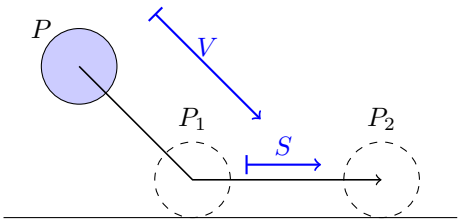


Figure 2: Collision and sliding

be stuck . The problem is that a stair is a wall but with a small height. However the physics engine does only detect there is a collision and it calculates the sliding velocity as if it was a wall. We must use another shape, which is the capsule. This shape has the particularity that its bottom is round (like an half sphere). When it collides against a stair, due to its round shape, the sliding velocity calculated will be different from sliding velocity of the rectangle shape, because the stair would be seen as a slope instead of a wall. As in figure 3, the sliding velocity of the box would be completely vertical. But because of the gravity, it would not move at all. However the sliding velocity of the capsule is like a slope, because it's the tangent of the half circle at the collision point. And as long as the slope is not too steep, the character would be able to climb it. No special scripting is needed. That's why the capsule shape is one popular way to solve this problem.

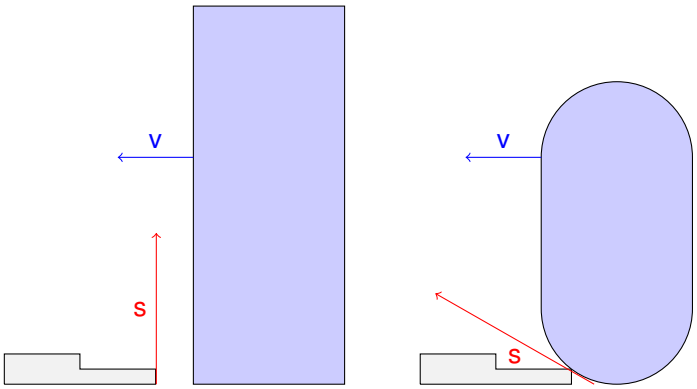


Figure 3: Problematic of the staircase

Legs But one constraint of this way is that the curve of the capsule must be very wide compared to the height of the stair. Practically it would be like the character could only climb up stairs of 1 cm high. Or the capsule must be very wide, meaning that our character is very fat. In both case, it's not very usable. Here comes the concept of leg. As in figure 4, the capsule is elevated and a ray is stuck below. The ray is the representation of the human leg. When we think about it, the reason why the character can't climb too high stairs is that the tangent of the collision point is too steep, and that's because the collision point on the curve of the capsule is too high. By elevating the capsule, the

collision point will be lower on the curve. And we use a ray for the leg because we need something to keep the capsule elevated but we don't want the leg to collide with the stair. The ray shape is special in the way that it has a width=0. It collides only on its length, here vertically.

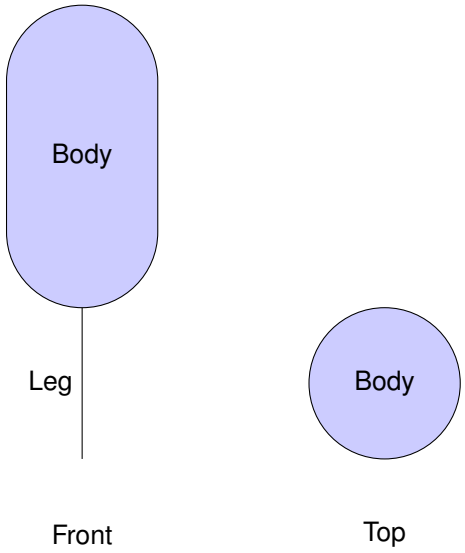


Figure 4: Skeleton of the kinematic body, version 1

7.2 Fly mode

Skeleton We will now implement a first version of this kinematic character. It is the simplest one, the fly mode. This implementation, as its name says, doesn't have gravity. It can just move in space and collide with objects, like a ship in space.

Let's start with the skeleton:

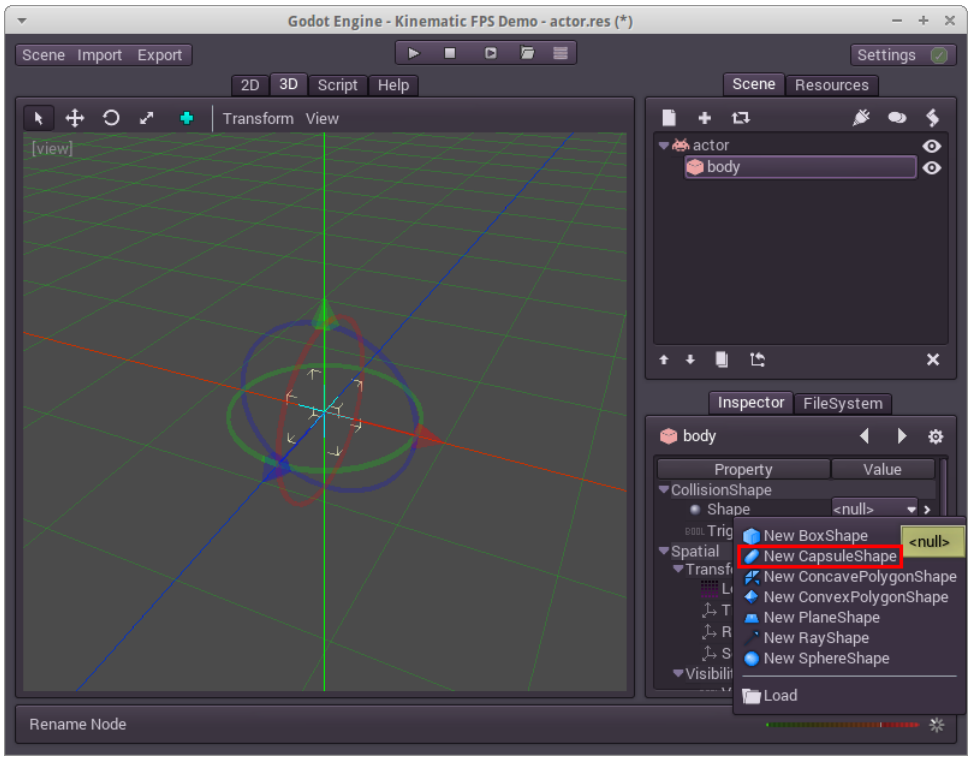
95) Create a new scene.

96) Add a *KinematicBody* node and rename it *actor*.

97) Save the scene as "actor.res" in the folder "res://".

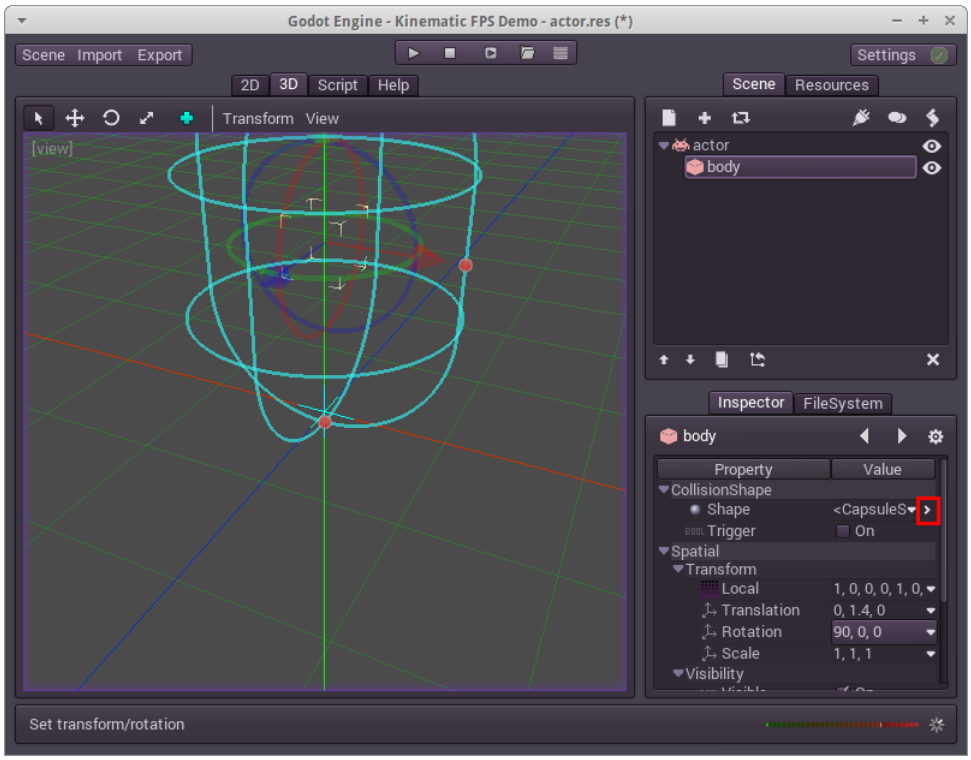
98) Select the *actor* node and add a *CollisionShape* node, named as *body*.

99) Select the *body* node and in its Shape property create a new Capsule-Shape.



100) Set its Translation to value [0,1.4,0] and its Rotation to [90,0,0].

101) Go in the detail of the property Shape.



102) Set its Radius to 0.6 and its Height to 0.8

103) Select the *body* node again.

104) Add another *CollisionShape* node and rename it *leg*.

105) For its shape, create a new RayShape.

106) Set its Translation to [0,0,1]

107) Go it the Shape detail and set value 0.4 for the Length property.

108) Select the *actor* node and create a new *Spatial* node. Rename it *yaw*.

109) With the *yaw* node selected, create a new *Camera* node. Rename it *camera*.

110) Set the Translation of *camera* to [0,1.7,0], and check the option Current.

Maybe you're wondering why I added a camera in an empty spatial node. The reason is that we want to keep the collision shapes as static, without rotation. Like in the old Mario game, the character itself doesn't rotate. This simpli-

fies a lot the calculations since a lot of variables are now constants, thanks to this arbitrary rule to not rotate.
The only nodes that rotates are the yaw, only for horizontal rotation, and the camera for vertical rotation. Why not both in one node, you ask? Again, it's to simplify the maths in the script to come. One big difference between 2D math and 3D math is that rotations act differently depending on which axis of rotation is used. By separating the camera rotation in two rotations, it is easier to understand which axis of rotation we must use.

Set up the main scene Since we have the node of the player, even thought it does nothing, we can test it already.

111) Open the "main.scn" scene

112) Select the *World* node

113) Instance the *actor* node we just created and rename it *player*

114) Move the node somewhere in the map. That will be its original position.

You can now give it a try and start the scene **[F5]**. You should normally see the map from where you put your *player* node.

Keys Before we continue, it is a good idea to configure now the keys we're going to use in this tutorial.

115) Open the project settings and go in Input map tab.

116) Add the following actions : move_forward, move_backward, move_left, move_right, jump, use

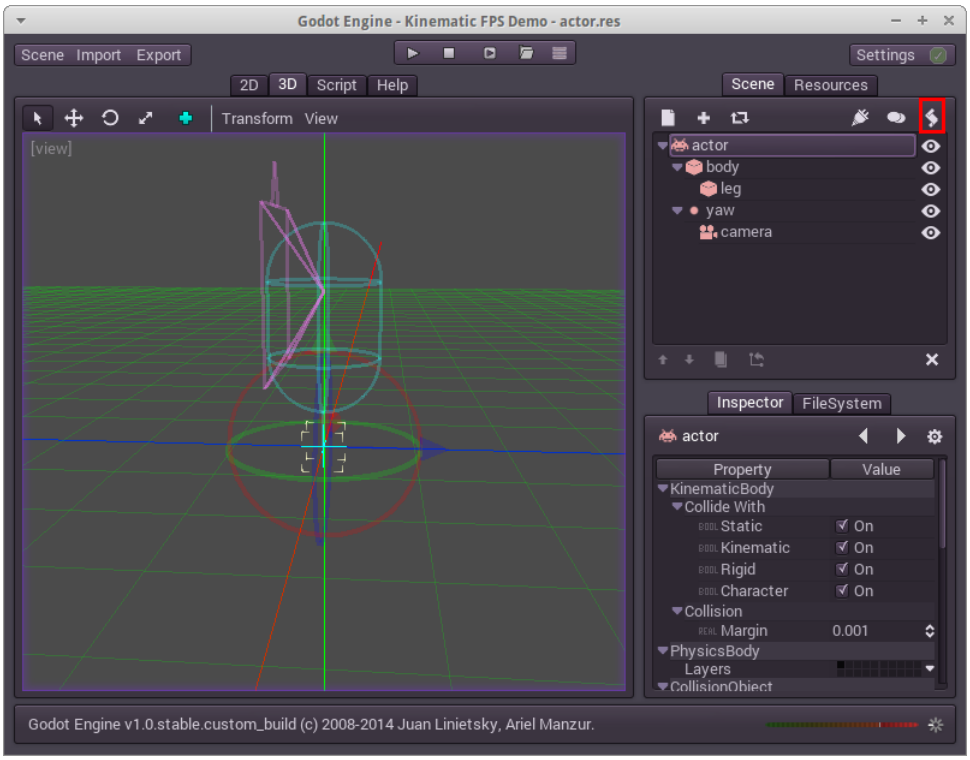
117) For each action, add the key you want. Personally I use the WASD configuration for movement, with space and E for jump and use.

118) Close the window.

Scripting

119) Open the "actor.res" scene.

120) Select *actor* node and click on the "Create the node script" button



121) Click ”Create” button

122) Replace the content of the script with this code:

```
extends KinematicBody

func _input(ie):
  pass

func _fixed_process(delta):
  pass

func _ready():
  set_fixed_process(true)
  set_process_input(true)
```

Hint

The process input function is called whenever a button is pressed or the cursor is moved. In our case, we’ll capture the mouse movement here.

Now that we have the most basic structure of the code, we’re going to implement the capture of the mouse movements.

123) At the beginning of the script, after the *extends KinematicBody* line, add those variables:

```
var view_sensitivity = 0.3
var yaw = 0
var pitch = 0
```

124) In the _input(ie) function, replace the content with this code:

```
if ie.type == InputEvent.MOUSE_MOTION:
  yaw = fmod(yaw - ie.relative_x * view_sensitivity, 360)
  pitch = max(min(pitch - ie.relative_y * view_sensitivity, 90), -90)
  get_node("yaw").set_rotation(Vector3(0, deg2rad(yaw), 0))
  get_node("yaw/camera").set_rotation(Vector3(deg2rad(pitch), 0, 0))
```

Hint

The variables *yaw* and *pitch* are the rotations of the camera. As explained before, we apply the *yaw* rotation to the spatial parent of the camera instead of the camera, as we do for the pitch. Also, since we capture the delta of the mouse movements, we must check that the yaw angle doesn't go too high or negative. Therefore we apply a modulo to this value. For the pitch, we put a limit instead so the camera's orientation is always with the up on the top of the screen. Since the actor's body never rotates vertically, the camera must have the same up orientation as the body.

125) Append at the end of the script this code:

```
func _enter_tree():
  Input.set_mouse_mode(Input.MOUSE_MODE_CAPTURED)

func _exit_tree():
  Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)
```

Hint

This makes the game capture completely the mouse. You won't even see cursor. But this is required to let the camera be able to move easily in every direction. It has however the problem that you can only quit the game with a hotkey (**[Alt-F4]** or a key you program yourself to quit the game). It is particularly annoying when Godot suddenly stops because of a script error, because there's no other way than try to kill the process to get the mouse back (and you have to do it without mouse!). Be sure to know the useful hotkeys of your OS before you try a Godot code that captures the mouse. For Ubuntu, **[Super-T]** opens a terminal, and you can find the process with ps and kill it. In Windows, **[Ctrl-Shift-Esc]** opens the task manager. But it's safer to disable the *set_mouse_mode(...)* code whenever you are not sure.

If you try to run the game now, you should be able to move the camera with the mouse in every direction. Take note that no matter how much you rotate, the ceiling is always above the screen and the floor is always below (except if you look at it directly).

Move the actor We can now focus on make the actor move when we press a key.
The algorithm is described in figure 5. Take note that in the fly mode, velocity is at every frame set to zero. Thus there's no inertia.
Also, it is not shown in the diagram, but because the slide function of Godot is not perfect at the time I'm writing the tutorial, the actual implementation tries a couple of times to slide the actor until it doesn't slide anymore. This trick is only needed in the fly mode.

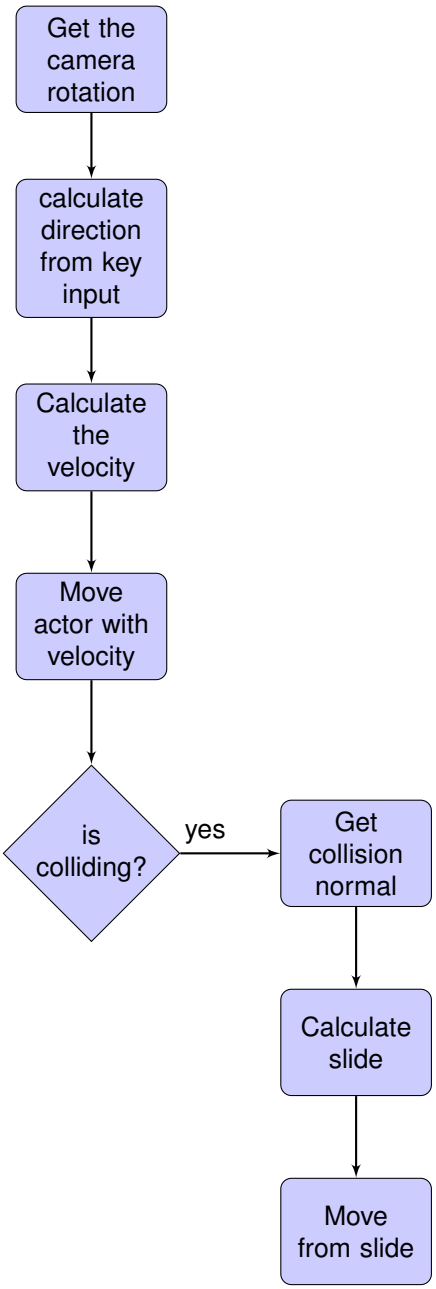


Figure 5: Algorithm of the fly mode

126) Add those variables and constants with the others variables:

```
var velocity=Vector3()

const FLY_SPEED=100
```

```
const FLY_ACCEL=4
```

127) Append at the end of the script:

```
func _fly(delta):
  # read the rotation of the camera
  var aim = get_node("yaw/camera").get_global_transform().basis
  # calculate the direction where the player want to move
  var direction = Vector3()
  if Input.is_action_pressed("move_forward"):
    direction -= aim[2]
  if Input.is_action_pressed("move_backward"):
    direction += aim[2]
  if Input.is_action_pressed("move_left"):
    direction -= aim[0]
  if Input.is_action_pressed("move_right"):
    direction += aim[0]

  direction = direction.normalized()

  # calculate the target where the player want to move
  var target=direction*FLY_SPEED

  # calculate the velocity to move the player toward the target
  velocity=Vector3().linear_interpolate(target,FLY_ACCEL*delta)

  # move the node
  var motion=velocity*delta
  motion=move(motion)

  # slide until it doesn't need to slide anymore,
  #or after n times
  var original_vel=velocity
  var attempts=4 # number of attempts to slide the node

  while(attempts and is_colliding()):
    var n=get_collision_normal()
    motion=n.slide(motion)
    velocity=n.slide(velocity)
    # check that the resulting velocity is not opposite to
    # the original velocity, which would mean moving backward.
    if(original_vel.dot(velocity)>0):
        motion=move(motion)
        if (motion.length()<0.001):
          break
    attempts-=1
```

Hint

You might have noticed there is a strange *Vector3().linear_interpolate* function call. In fact, the *Vector3()* is normally the old velocity from last frame. But since we don't want to keep the velocity in fly mode, I replaced it with a new empty vector, meaning no velocity.

Hint

Another interesting line to understand is the *get_node("yaw/camera").get_global_transform().basis*.
More detail in the annexe chapter 12.1 about camera aiming.

128) And finally replace the content of function *_fixed_process* with:

```
_fly(delta)
```

If you run now the game, you should be able to move in your map.

Hint

You can find the complete script and scene for the actor at this point in the project's folder "tutorial/fly_mode".

7.3 Walk mode

129) Before we go further, make a copy of the *_fly* function and rename the copy *_walk*.

Important!

Don't change directly the *_fly*function, we'll need it later.

130) change the content of *_fixed_process* to :

```
_walk(delta)
```

7.3.1 Velocity

131) In the fields/variables section, add the following code:

```
const WALK_MAX_SPEED = 15
const ACCEL= 2
const DEACCEL= 4

var is_moving=false
```

132) At the beginning of the *_walk* function, just before *direction = direction.normalized()*, add:

```
#reset the flag for actora's movement state
is_moving=(direction.length()>0)
```

133) Replace this line:

```
var target=direction*FLY_SPEED
```

with

```
var target=direction*WALK_MAX_SPEED
# if the character is moving, he must accelerate.
# Otherwise he deccelerates.
var accel=DEACCEL
if is_moving:
  accel=ACCEL
```

134) Replace this line:

```
# calculate the velocity to move the player toward the target
velocity=Vector3().linear_interpolate(target,FLY_ACCEL*delta)
```

with

```
# calculate velocity's change
var hvel=velocity
hvel.y=0

# calculate the velocity to move toward the target,
# but only on the horizontal plane XZ
hvel=hvel.linear_interpolate(target,accel*delta)
velocity.x=hvel.x
velocity.z=hvel.z
```

Give it a try and run the game. ...Hey! We can't climb up in the air anymore! It's like we're stuck on an imaginary floor.

That's right. The changes we did just now added inertia to our velocity (you've maybe noticed that when you stop walking, you slow down first) but only on the horizontal plane. We nullified the Y value in the direction we try to go. That's because when we walk, we're stuck to the ground thanks to gravity. We can then consider that we have only the ability to move on a 2D plan, jumping being an exception. It looks weird now, but once we implement gravity it will make more sense.

Hint

We added at the same time the variable *is_moving*. It will have multiple uses later, but now it helps to know if the actor must accelerate or decelerate. Because in video game physics, friction can be simplified by considering that there's friction only when something is not moving by itself. And when it wants to move, it accelerates before it reaches the maximum speed.

Hint

Did you also noticed that the speed changed? And on top of that it seems there is a maximum speed we can't go beyond. That's thanks to the *linear_interpolate* function. This function calculates a vector based on another vector and a length. If the length is bigger than the reference vector's length, it will be capped to the reference vector's length.

7.3.2 Gravity

135) Add this code to the fields/variables section:

```
const GRAVITY=-9.8*3
```

Hint

Gravity is multiplied 3 times because it gives a better feeling. I'm not sure why, but it's probably related to a unit conversion problem (9.81 is the gravity acceleration constant in $m * s^2$, but there's no meter unit in Godot). You can change it to your content.

136) After *direction = direction.normalized()*, add:

```
# add gravity
velocity.y+=delta*GRAVITY
```

If you test it now, you'll see yourself falling to the ground. And if you go on a slope, you slide down to the bottom, no matter how steep the slope is. However you can see that depending on how steep the slope is, you have more or less difficulty to climb it. And check for the stairs. If you made a staircase that have low stairs, you'll be able to climb up the staircase, but not easily. That's thanks to our capsule shape.

Congratulation! You made your very first FPS character. It's from far not perfect, but it walks on the ground and it falls. It can even climb slopes and stairs (somehow).

Now we're going to fix all those problems.

Hint

You can find the complete script and scene for the actor at this point in the project's folder "tutorial/basic_gravity".

7.3.3 Jump

Jumping is a simple matter of giving a vertical impulse.

137) Add in the field section:

```
const JUMP_SPEED = 3*3
```

138) Add at the end of the function:

```
if Input.is_action_pressed("jump"):
  velocity.y=JUMP_SPEED
```

If you try it, you see that you can now jump vertically. But as long as you press the jump key, you jump. Even in the middle of the air. We need to add a condition to allow to jump only when we're on the ground. But this comes in the next chapter.

7.3.4 Ground-clamping

If you played a bit with the current state of the game, you might have noticed a couple of problems:

- You can't stay on a slope, no matter how low it is. It's like you're standing on ice.
- Jumping should be allowed only when you're on floor.
- When you walk on a slope from the top, you're jumping first before you slide down.

All issues point to the need to detect when the character touches the floor. Not only but also the last issue is a bit more complicated. In figure 6, you see the actor on a slope. If the actor has a velocity, it's a horizontal velocity. It means that when the actor moves, he will lose foot for an instant, and then he'll fall due to gravity (in red). If the velocity is small, it is unnoticeable, especially now that the actor can't even stand still on the slope. But if he goes faster, he'll be constantly falling, as if he jumps down. What we want instead is that the actor follows the ground (in blue). But for that, we need to make the actor to stick to the ground. This is what we call ground clamping.

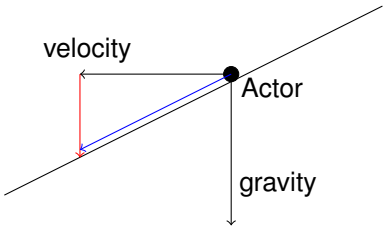


Figure 6: Problem of sliding down a slope from a side view

Raycast The approach to solve those problems is to add a *raycast* below the player like in figure 7, and if a collision is detected, teleport the player to the collision point.

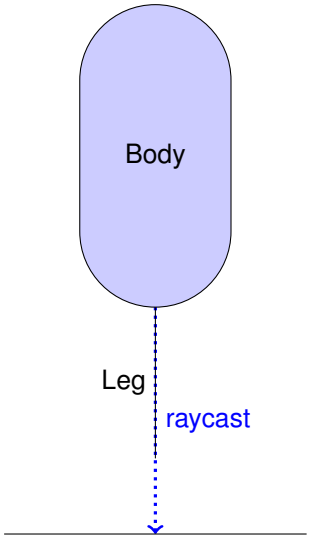


Figure 7: Skeleton of the kinematic body, version 2

The length of the ray is quite important, because on a slope it will be the tolerance margin to consider that the actor is still on the ground, as shown in figure 8. If the ray is too short and the velocity is too high, the actor would be in the air the moment it reaches a slope, then fall.
But on the other side the ray must not be too long, because it would be like the actor is always touching the ground.

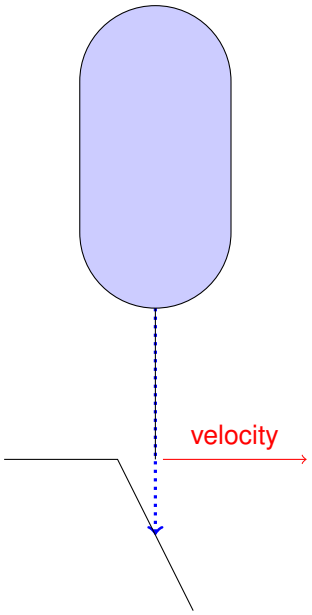


Figure 8: Margin of the raycast on a slope

Implementation

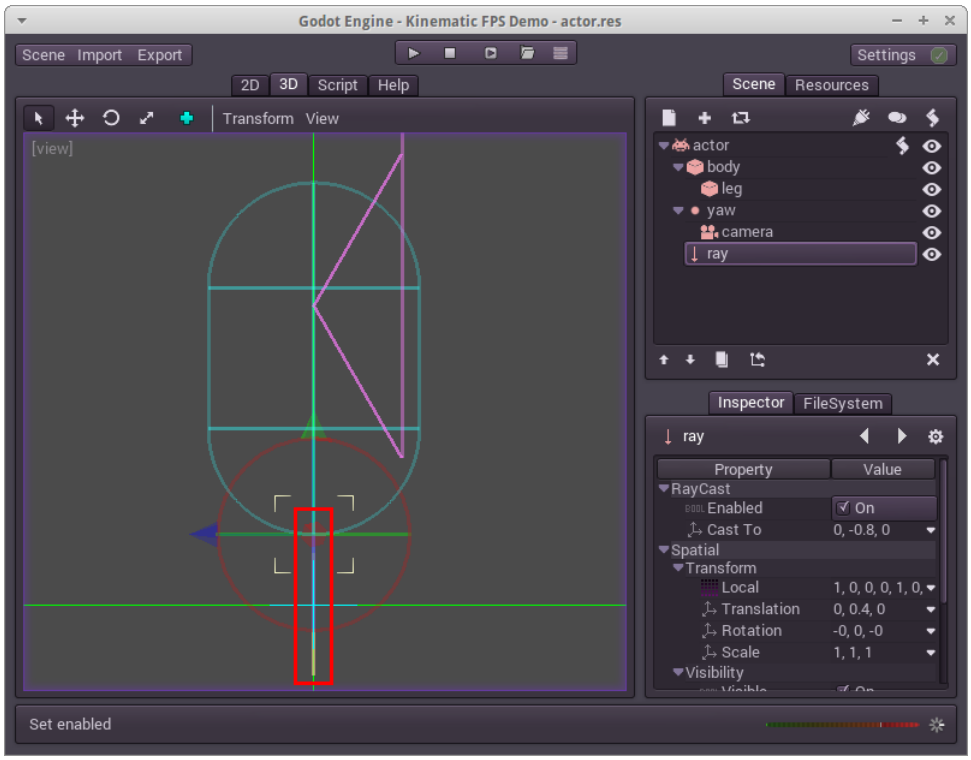
139) In the scene tree of the actor, select the *actor* node.

140) Add a *RayCast* node (don't confuse it with *RayCast2D*) and name it *ray*.

141) Move it to *Translation* [0,0.4,0], just at the intersection between *body* and *leg* nodes.

142) Check property *Enabled*

143) Set the property *Cast To* to [0,-0.8,0]
You should now have a scene structure like this:



It's hard to see where the *ray* node is, but it's in the red rectangle I've drawn.

144) In the script file, add in the fields section:

```
var on_floor=false
```

145) Add at the very beginning of the *_walk* function:

```
var ray = get_node("ray")
```

146) Replace the code:

```
direction = direction.normalized()
```

with:

```
direction.y=0
direction = direction.normalized()

# clamp to ground if not jumping. Check only the first time a
# collision is detected (landing from a fall)
var is_ray_colliding=ray.is_colliding()
```

Hint

For convenience we nullify right here the Y of the direction. We are sure now that the actor moves only horizontally.

147) Right after that, add the following code:

```
if !on_floor and is_ray_colliding:
  set_translation(ray.get_collision_point())
  on_floor=true
elif on_floor and not is_ray_colliding:
  # check that flag on_floor still reflects the state of the ray.
  on_floor=false
```

Hint

Here we did 2 things. Not only we clamp to the ground the actor when he detected the collision, which we know it's the ground since it's right below, but we also configured a flag(field) *on_floor* to whether if the actor is on the floor or in air. You might have noticed that we teleport the actor only if there is a change between the collision state of the ray and the flag. By not constantly teleport the actor to the ground, not only we avoid useless code that would only slow down (a little) the game, but it avoids also possible bugs of the player moving vertically for no reason (for instance in an elevator).

148) Replace the code:

```
# add gravity
velocity.y+=delta*GRAVITY
```

with:

```
if on_floor:
  pass
else:
  # apply gravity if falling
  velocity.y+=delta*GRAVITY
```

Hint

Gravity is now applied only when the actor is in the air. He doesn't slide anymore when he's on a slope. But it doesn't consider the angle of the slope for the moment.

149) We can simplify the motion part. Replace:

```
var attempts=4 # number of attempts to slide the node

while(attempts and is_colliding()):
  var n=get_collision_normal()
  motion=n.slide(motion)
  velocity=n.slide(velocity)
  # check that the resulting velocity is not opposite to the
  # original velocity, which would mean moving backward.
  if(original_vel.dot(velocity)>0):
    motion=move(motion)
    if (motion.length()<0.001):
      break
  attempts-=1
```

with:

```
if(motion.length()>0 and is_colliding()):
  var n=get_collision_normal()
  motion=n.slide(motion)
  velocity=n.slide(velocity)
  # check that the resulting velocity is not opposite to the
  # original velocity, which would mean moving backward.
  if(original_vel.dot(velocity)>0):
    motion=move(motion)
```

Hint

It does the same thing as before, but because our way to calculate the velocity changed, I noticed with practice that multiple move attempts are not needed anymore. I've then only removed everything that became useless and that would only slow the game down.
And I added a condition. If the actor doesn't move (*motion.length==0*), no need to do all this extra calculation.

150) Let's finish with the jump. Replace:

```
if Input.is_action_pressed("jump"):
 velocity.y=JUMP_SPEED
```

with:

```
if on_floor:
  if Input.is_action_pressed("jump"):
    velocity.y=JUMP_SPEED
```

Now give it a try. We can't fly with the jump key anymore. And we can stand on a slope, though it's not perfect and we still slide slowly.
However new bugs appeared. When you climb a slope and stop, you're elevating in the air before you suddenly teleport to the ground. And when you going down the slope, you're first falling before you slide down. The reason it that the gravity is not applied all the time anymore and the vertical velocity is conserved because no friction is applied to it. So the actor goes up until the ray doesn't detect the ground anymore, then it falls and loses its vertical velocity.

It's definitely not perfect. But we'll fix all this.

Hint

You can find the complete script and scene for the actor at this point in the project's folder "tutorial/ground_clamping".

7.3.5 Slopes

In the previous section, what is missing is the part to manage the vertical velocity. When you move, the X and Z of the actor's velocity accelerate. And when you touch nothing, the X and Z of the actor's velocity decelerate. But in a slope, the Y changes too due to the *slide* function. However, when you don't move anymore, the Y of the velocity doesn't decelerate and the actor starts to elevate until it reaches the point where the ray doesn't detect the ground anymore. And then the actor falls and the Y axis of its velocity is set to 0.
In figure 9, step 1. shows the actor moving with a velocity.
Step 2. show the actor when the player doesn't press a movement key; the X velocity is decelerated to 0 but the remaining Y velocity is still there.
Step 3. shows the actor in the air and its ray that doesn't touch the ground anymore. Gravity applies again and the actor starts to fall.

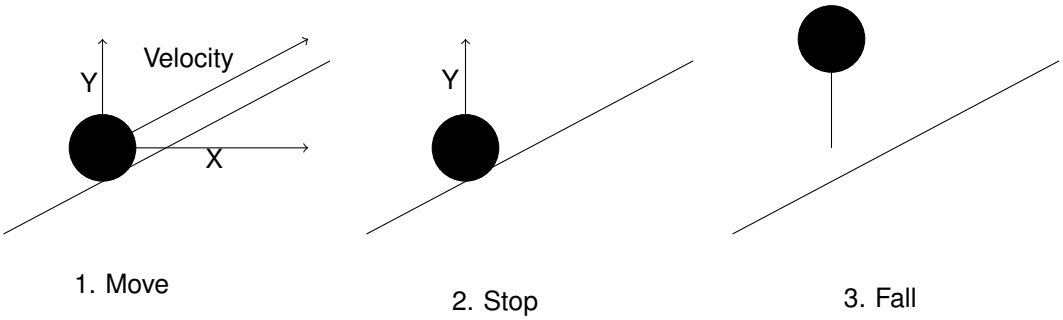


Figure 9: Problem of Y velocity when climbing the slope

As well, when running down a slope, the actor starts by walking horizontally until the ray doesn't touch the ground anymore and then starts to fall and follow the slope because of the gravity. That's because as long as the actor's ray touches the ground and the flag *on_floor* is true, the actor thinks he's on the

floor and doesn't need to apply gravity.
To fix that, we need to adapt the velocity to the floor. As you can see in figure 10, the actor has a remaining velocity vel_0 from previous frame. The actor collides with the ground thanks to the raycast, and the normal of the ground at the collision point is known as $normal$. By projecting vel_0 to the ground, perpendicularly to $normal$, we can get a velocity vel_1 that impacts the Y value of the original velocity. And this will solve both issues explained before, since the Y value of velocity will be indirectly decelerated.

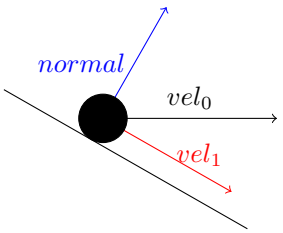


Figure 10: Project the velocity to the ground

To do that, we use a scalar projection based on the normal of the ground. More explanations about scalar projection at chapter 12.2.

151) In the script file, add in the fields section:

```
const MAX_SLOPE_ANGLE = 40
```

152) Replace the code:

```
if on_floor:
  pass
else:
  # apply gravity if falling
  velocity.y+=delta*GRAVITY
```

with:

```
if on_floor:
  # if on floor move along the floor. To do so, we calculate the
  # velocity perpendicular to the normal of the floor.
  var n=ray.get_collision_normal()
  velocity=velocity-velocity.dot(n)*n

  # apply gravity if on a slope too steep
  if (rad2deg(acos(n.dot(Vector3(0,1,0))))> MAX_SLOPE_ANGLE):
    velocity.y+=delta*GRAVITY
else:
```

```
  # apply gravity if falling
  velocity.y+=delta*GRAVITY
```

Give it a try.

That's quite good! Now we're totally sticking to the ground. And we don't slide at all on the slope that's not too steep, as well as we can't climb the other slope that is too steep. In fact, we stick so well to the ground that we can't even jump anymore, nor climb stairs (or very hardly). That's because the jump impulse we give is transformed to be parallel to the ground, which this time shouldn't be the case. But we are in the right direction. We just need to implement a couple of tricks.

Hint

You can find the complete script and scene for the actor at this point in the project's folder "tutorial/slopes".

7.3.6 Jumping again

Jumping is not possible anymore because the ground clamping works too well. One simple solution for that is to temporarily disable the ground clamping to give to the actor the time to jump.

153) In the script file, add in the fields section:

```
var jump_timeout=0

const MAX_JUMP_TIMEOUT=0.2
```

154) At the beginning of the *_walk* function, add:

```
# process timers
if jump_timeout>0:
  jump_timeout-=delta
```

155) Replace the code:

```
  if !on_floor and is_ray_colliding:
```

with:

```
  if !on_floor and jump_timeout<=0 and is_ray_colliding:
```

156) Replace the code:

```
if Input.is_action_pressed("jump"):
  velocity.y=JUMP_SPEED
```

with:

```
if Input.is_action_pressed("jump"):
  velocity.y=JUMP_SPEED
  jump_timeout=MAX_JUMP_TIMEOUT
  on_floor=false
```

Try it. We can now jump again. What's left are the stairs.

Hint

You can find the complete script and scene for the actor at this point in the project's folder "tutorial/jump_again".

7.3.7 Stairs

For the problem of the staircase we'll use another trick. We'll use another raycast to detect stairs. If a stair is detected and if it's at the direction the actor is trying to move, we'll make the actor automatically jump. The concept is as described in figure 11 by putting a raycast in front of the actor. This step raycast must not collide with the ground and thus must be placed higher than the *leg*. This raycast will be only used when the player moves and it will be positioned always where the actor is going, meaning the velocity's direction.

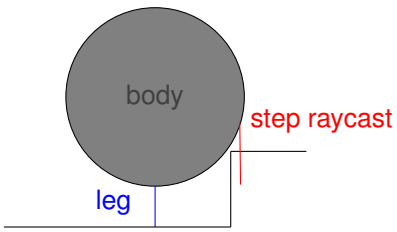


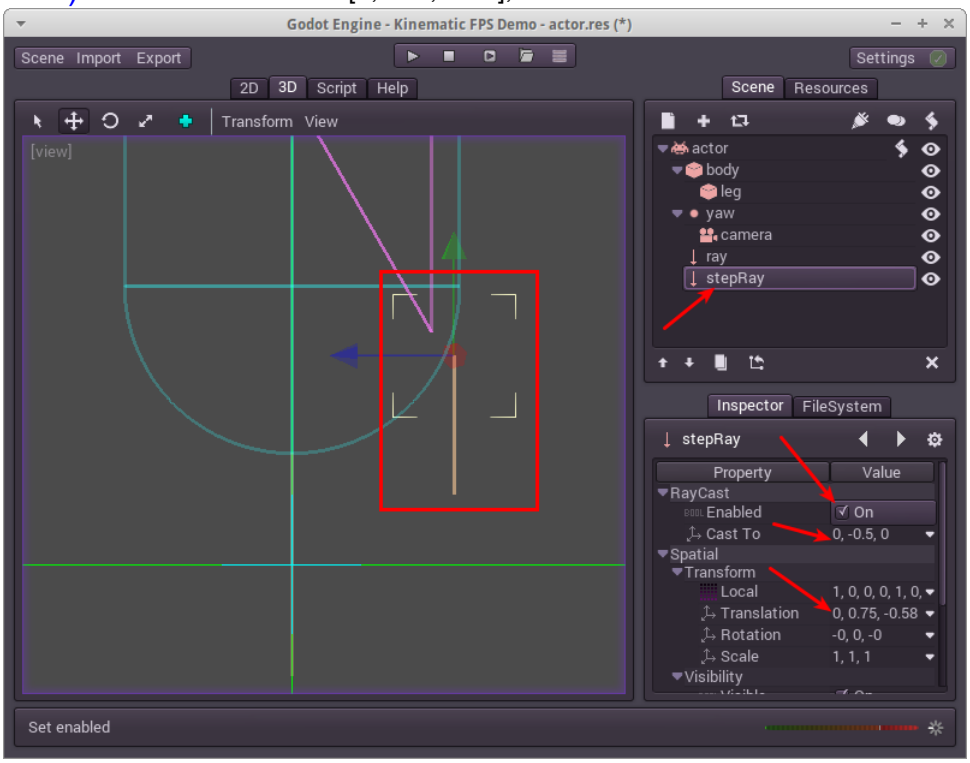
Figure 11: Side view of the step raycast detecting a stair

First we complete the player's skeleton:

157) In the scene tree, select the *actor* node and add a *RayCast* node. Call it *stepRay*.

158) Mark it *Enabled* and give it a *Cast To* to [0,-0.5,0]

159) Move it to *Translation* [0,0.75,-0.58], like this:



Then we implement the script:

160) In the script file, add in the fields section:

```
const STAIR_RAYCAST_HEIGHT=0.75
const STAIR_RAYCAST_DISTANCE=0.58
const STAIR_JUMP_SPEED=5
const STAIR_JUMP_TIMEOUT=0.2
```

161) In function *_walk*, after the line:

```
var ray = get_node("ray")
```

add the line:

```
var step_ray=get_node("stepRay")
```

162) Replace the code:

```
if on_floor:
  # if on floor move along the floor. To do so, we calculate the
```

```
  # velocity perpendicular to the normal of the floor.
  var n=ray.get_collision_normal()
  velocity=velocity-velocity.dot(n)*n

  # apply gravity if on a slope too steep
  if (rad2deg(acos(n.dot(Vector3(0,1,0))))> MAX_SLOPE_ANGLE):
    velocity.y+=delta*GRAVITY
else:
  # apply gravity if falling
  velocity.y+=delta*GRAVITY
```

with:

```
if on_floor:
  # if on floor move along the floor. To do so, we calculate the
  # velocity perpendicular to the normal of the floor.
  var n=ray.get_collision_normal()
  velocity=velocity-velocity.dot(n)*n

  # if the character is in front of a stair, and if the step is
  # flat enough, jump to the step.
  if is_moving and step_ray.is_colliding():
    var step_normal=step_ray.get_collision_normal()
    if (rad2deg(acos(step_normal.dot(Vector3(0,1,0))))< MAX_SLOPE_ANGLE):
      velocity.y=STAIR_JUMP_SPEED
      jump_timeout=STAIR_JUMP_TIMEOUT

  # apply gravity if on a slope too steep
  if (rad2deg(acos(n.dot(Vector3(0,1,0))))> MAX_SLOPE_ANGLE):
    velocity.y+=delta*GRAVITY
else:
  # apply gravity if falling
  velocity.y+=delta*GRAVITY
```

163) And finally, add at the end of the function:

```
# update the position of the raycast for stairs to where the
# character is trying to go, so it will cast the ray at the next
# loop.
if is_moving:
  var sensor_position=Vector3(direction.z,0,-direction.x)*STAIR_RAYCAST_DISTANCE
  sensor_position.y=STAIR_RAYCAST_HEIGHT
  step_ray.set_translation(sensor_position)
```

Hint

This last code calculates a position from the resulting velocity of this frame step, and then set the *stepRay* at this position. Since the ray is a child of the *actor*, its position is relative.

Let's try it.

Much better! We can now climb stairs. It's not very fluid. If you compare with the old Doom[9] game, the player in Doom could climb stairs without losing velocity, where here the actor is more or less slowed down depending on the stairs height. But the way Doom was made is completely different from actual physics like the one Godot uses, and it wasn't exactly realistic anyway. For this tutorial, it will be more than enough.

That's it. We finished the implementation of a kinematic FPS character. You could create a map and put this node in it, and you'll be able to visit this map. However, the tutorial doesn't end here. Everything is fine, but only as long as the character doesn't interact with moving objects nor actionable objects (buttons, doors, ladders). The next chapters will cover those special aspects.

Hint

You can find the complete script and scene for the actor at this point in the project's folder "tutorial/climbing_stairs".

8 Elevators

We are now going to make things move in this map. The simplest ones are the elevators and moving platforms. Alas, they are also not the most intuitive (for the moment) to animate with our actor implementation.

Elevators are imported as meshes that are part of the map but still are separate objects. Normally you should see in your scene in Godot that elevators were imported as separate nodes.

Animation In Godot, you can animate a node very easily thanks to a special node *AnimationPlayer*. This allows you to change attributes, including *Translation*, in a timeline that you define. The most obvious and easiest way to move an elevator from point A to point B is to set the *Translation* property to both points at desired time. However if an object stands on this elevator, it won't move automatically with the elevator. For instance, if the player stands on a platform that moves horizontally, the platform would move while the player stand still. And eventually the player will fall of the platform. The problem is that no friction is applied to objects on top of the moving platform. Only when the elevator moves only vertically can objects follow the elevator because of the gravity. Otherwise they must be moving themselves with the elevator.

Linear velocity At the moment I'm writing this tutorial, I still didn't find a way to calculate the real velocity of an elevator when it's animated with an *AnimationPlayer*. However, one of the demos I based my work on, called FPS Test [4],

used the *Angular Velocity* property of a *RigidBody*. This property makes the object rotate on itself at specified speed, without using any *AnimationPlayer*. Beside this property, there's the *Linear Velocity* property which moves the object the same way as the *Angular Velocity* rotates the object. If you set a non null vector to it, the body moves exactly at the speed of this vector. And since this vector can be accessed by others nodes, the trick when the player stands on the object is to read this vector and make the player move with the same velocity, on top of its own velocity but without changing it.

The problem is that makes more complicated to configure the elevator. You can with the *AnimationPlayer* change the *Linear Velocity* at any time with the desired velocity, but you have to calculated the correct velocity to make the platform arrive on time at destination, not too late nor too soon. And most of the time you'll have to find with lot of tries the right velocity. Probably someone will find a more convenient solution soon.

Erratum A solution has been found. You can animate an elevator with only the *Translation*, as long as the *Mode* is *Kinematic*. But follow the tutorial for the moment.

8.1 Preparation of the elevator node

When the map was imported, the evelators, as well as all others meshes, were imported in 3 nodes like in figure 12.

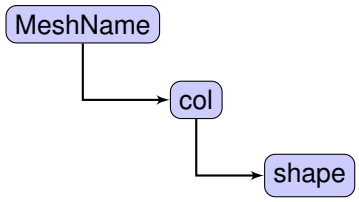


Figure 12: Elevator node structure

We want to swap the mesh and *col* nodes, because the *Linear Velocity* property is in *col*. At the end, we want a structure like figure 13.

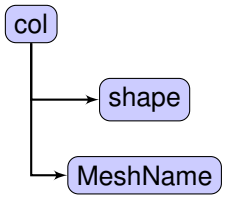


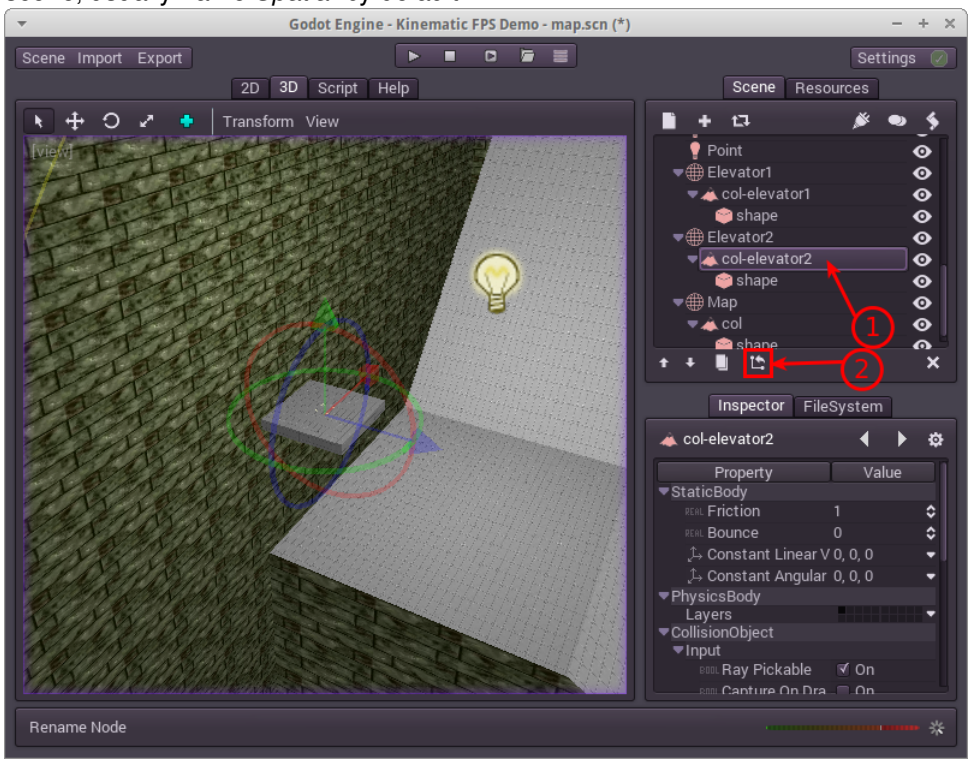
Figure 13: Corrected node structure

Erratum Changing the order of the nodes isn't necessary anymore. You can skip the steps about reparenting if you want.

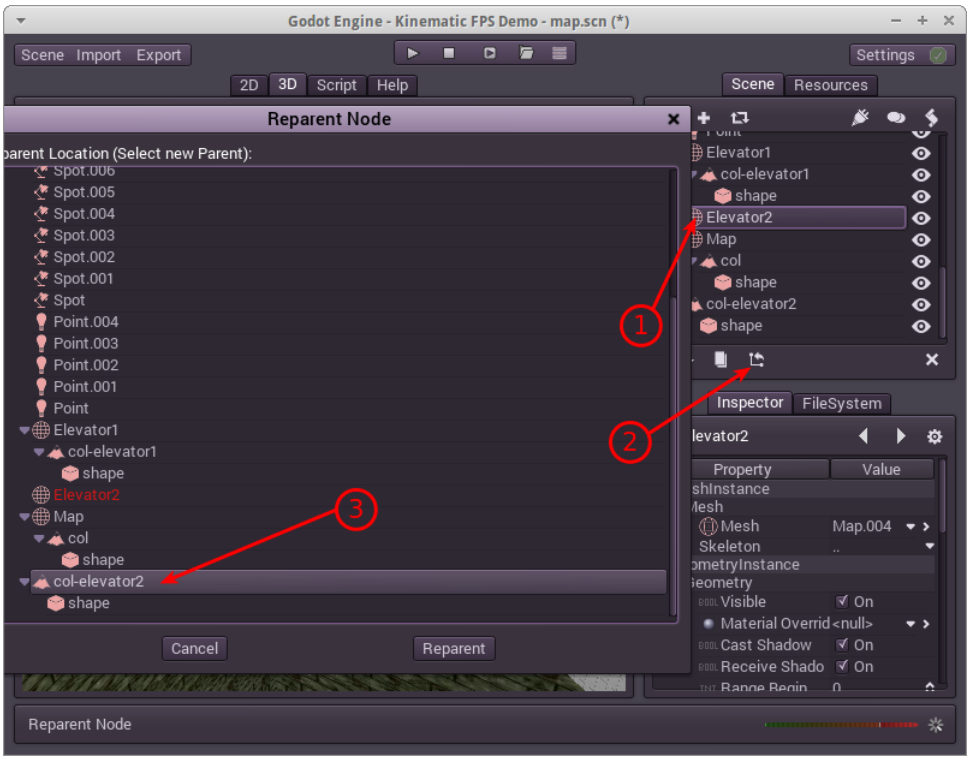
164) Open the map scene

165) Search for one elevator's node and select its *col* node. Rename it a unique name related to the elevator's name, for instance *col-elevator2* (because my mesh was named elevator2).

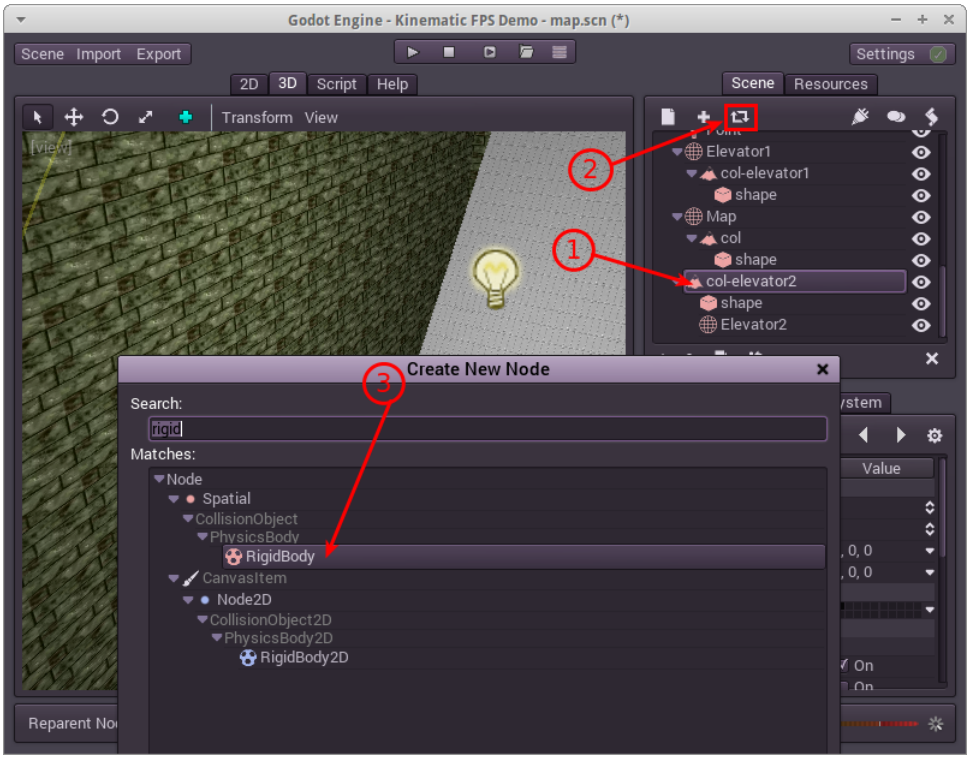
166) With the node still selected(1), reparent (2) it to the root node of the scene, usually name *Spatial* by default.



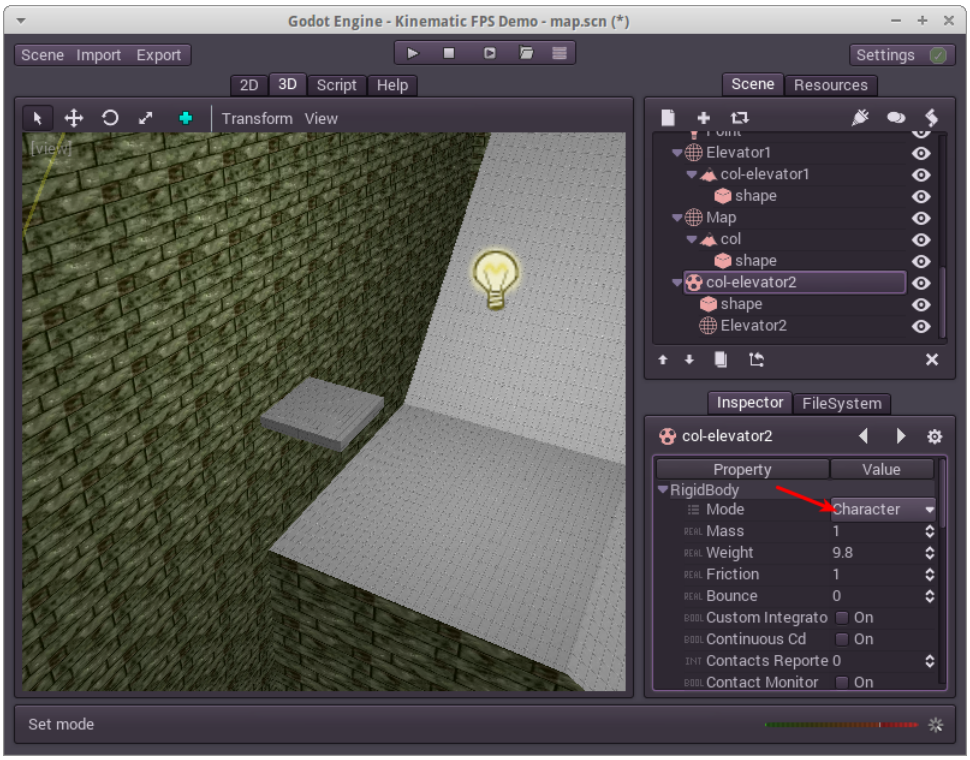
167) Select now the elevator node (1) and reparent (2) it to the node *col-elevator2* (3).



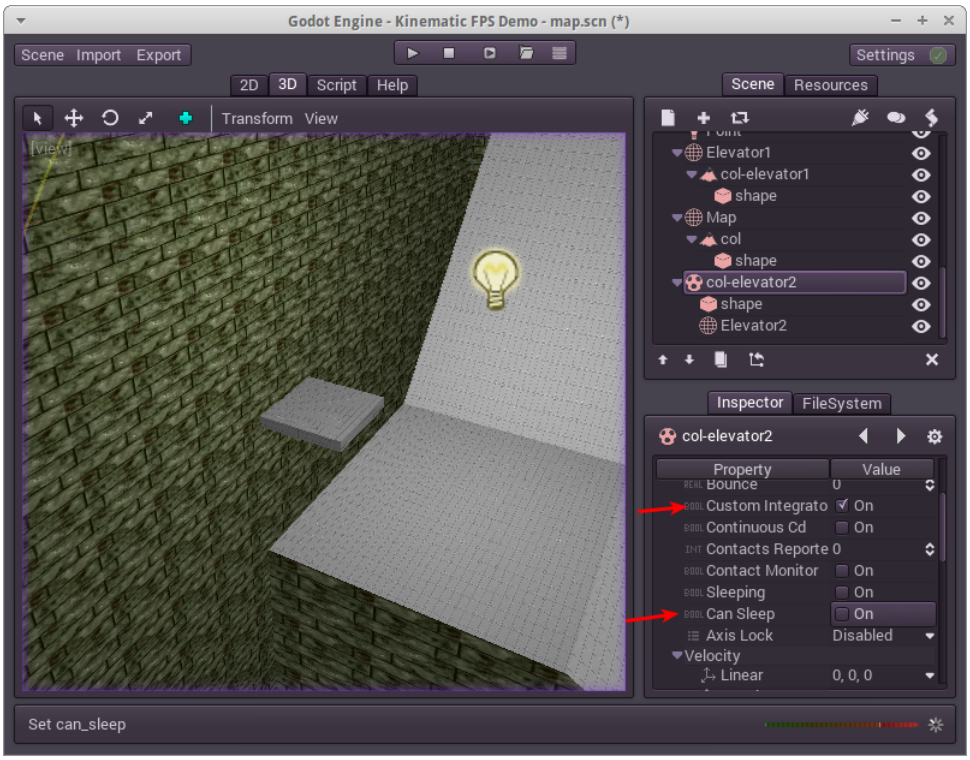
168) Select now the *col-elevator1* (1) and change its type(2) for a *RigidBody* (3).



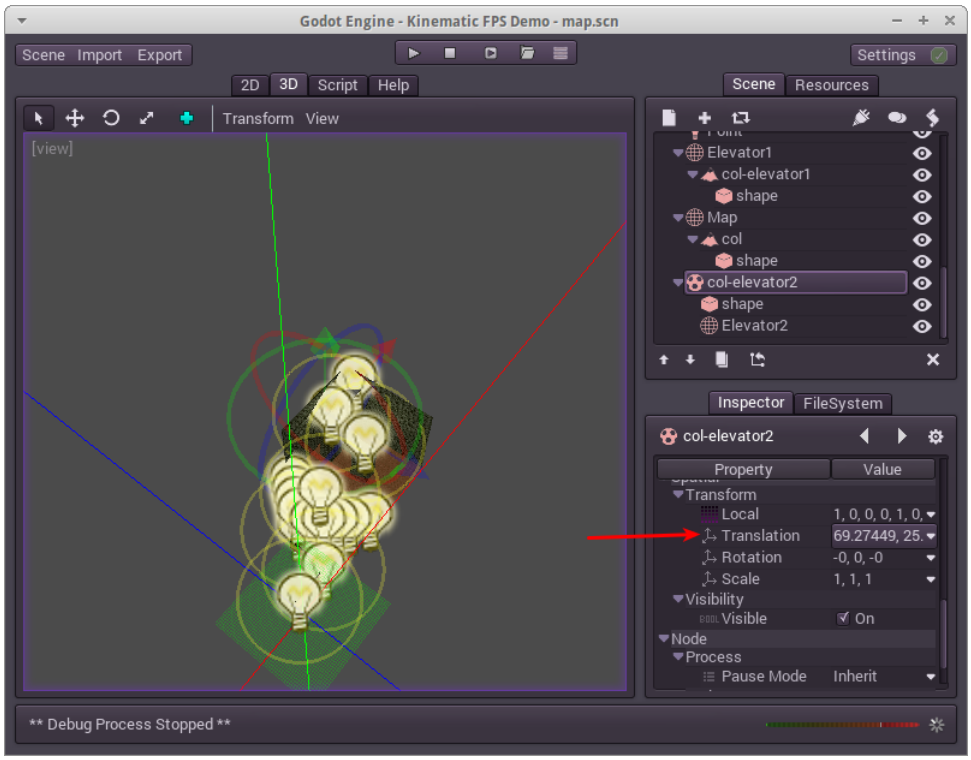
169) Change the *Mode* property to *Character*.
Erratum Set it to *Kinematic* instead.



170) Check *Custom Integration* and uncheck *Can sleep*.



171) Finally, copy the *Translation* values from the elevator node to the *col-elevator1* node. Then set the *Translation* of the elevator node to [0,0,0]. It's basically a swap of the *Translation* property between both nodes.



172) Repeat all those steps for all moving platforms and elevators.

Hint

I can't give much information about those options. In a nutshell, for what I've understood, the *Custom Integration* and *can sleep* are changed so the object doesn't fall and applies constantly its *Linear Velocity* and *Angular Velocity*.

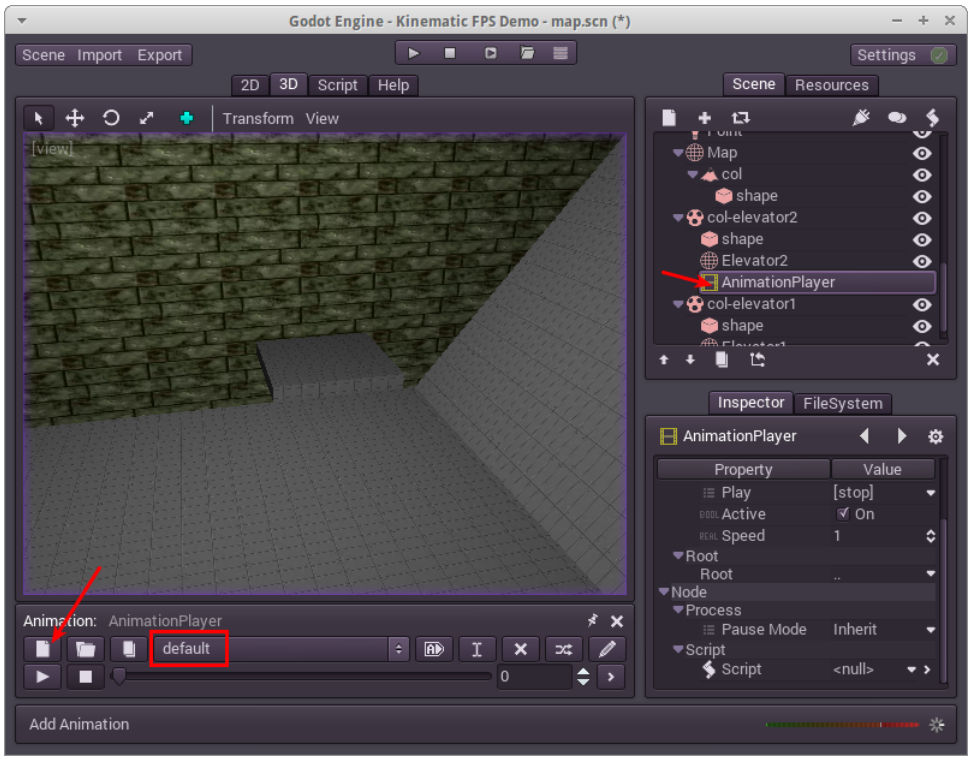
Test the scene before you go further, just to be sure that the elevator is still at its original place and that you can still jump on it without falling through.

8.2 Animation of the elevator

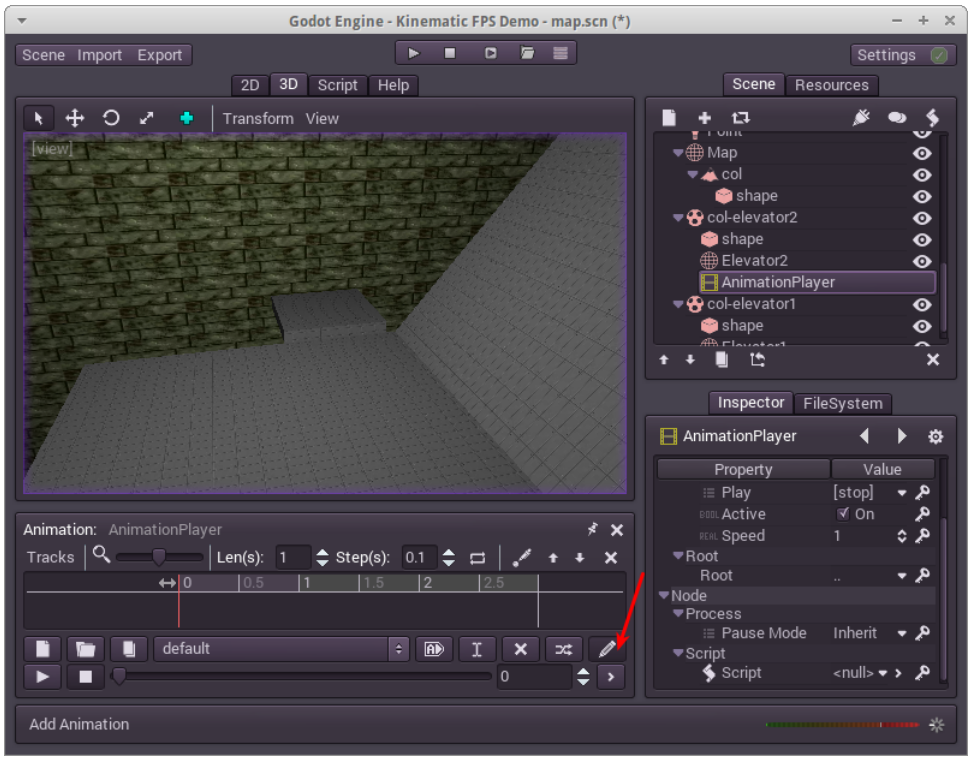
Erratum Before you do the next steps, if the rigid body is set to mode *Kinematic*, you don't need to set a linear velocity. You can simply animate the *Translation* instead.

173) Select *col-elevator1* node and add an *AnimationPlayer* node.

174) Select it and create a new animation, with the name for instance "default" (doesn't matter).

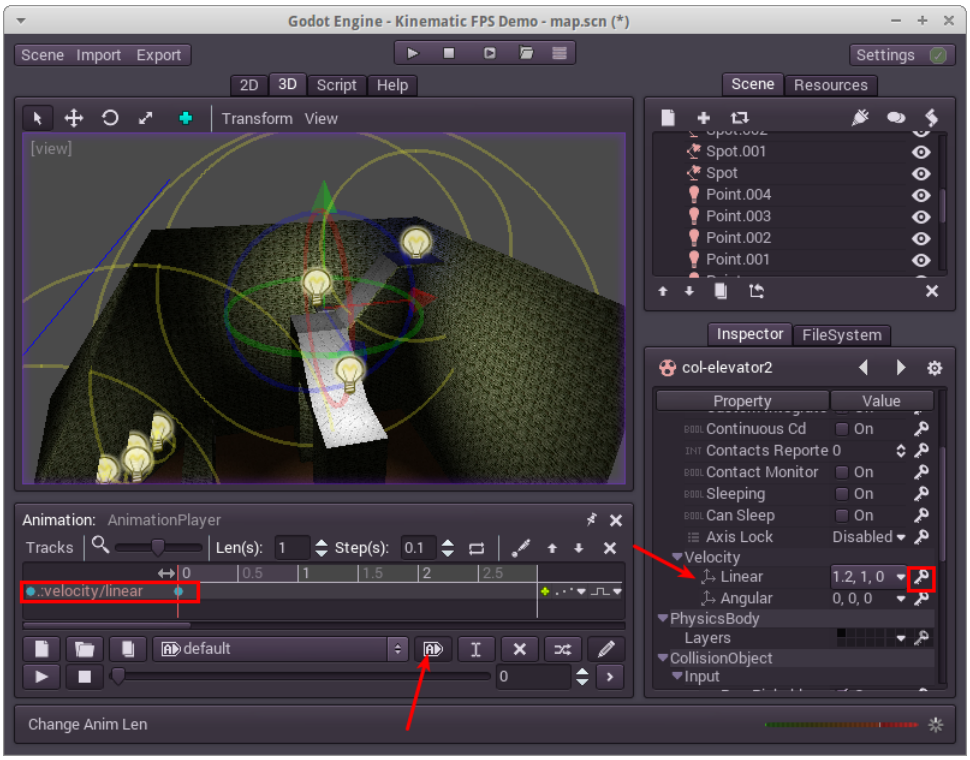


175) Open the animation editor

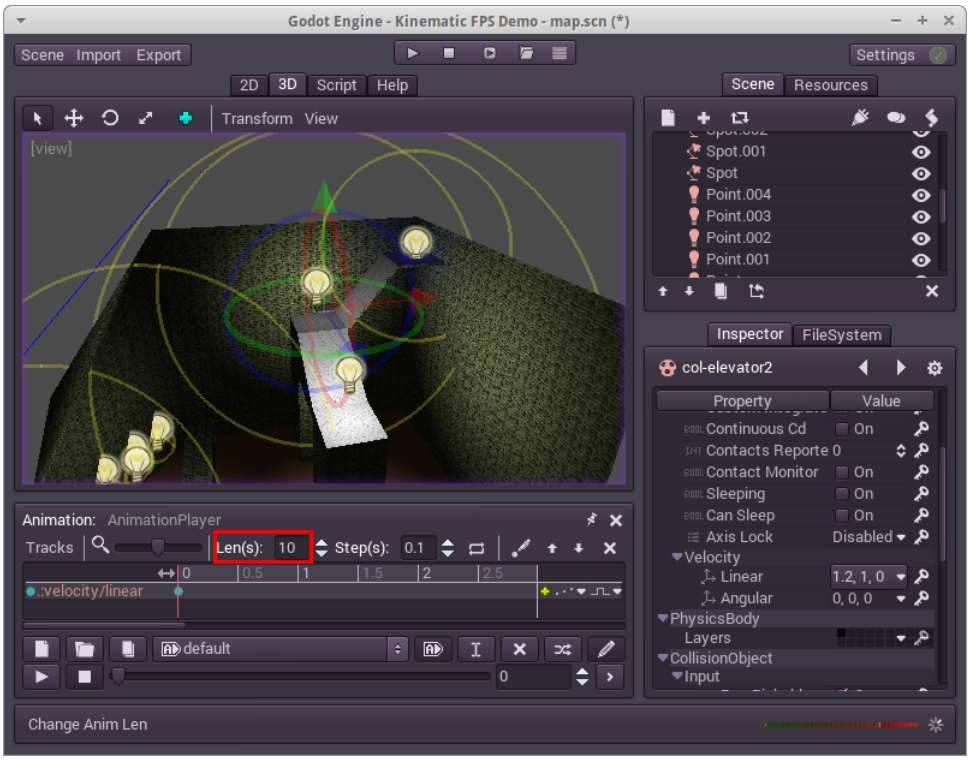


176) Select now the *colo-elevator2* node.

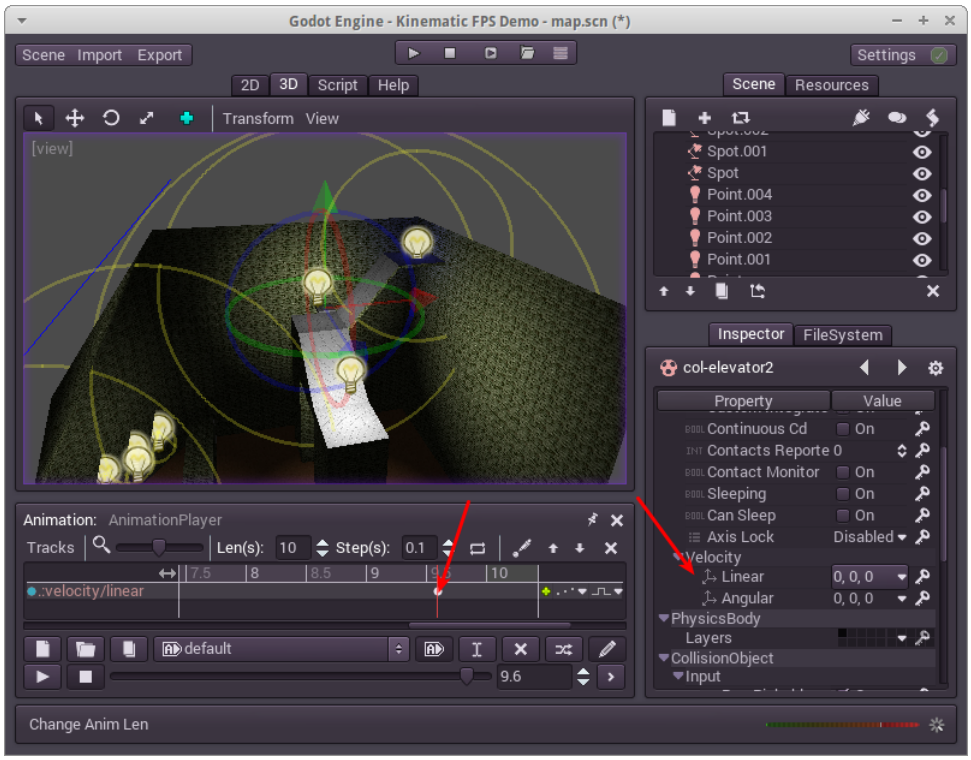
177) Then change its *Linear Velocity* to the desired velocity. For my elevator I used [1.2,1,0]. And click on the key to create an animation key. And while you're at it, click on the autoplay button, so that the animation starts automatically.



178) Change the length of your animation. For my elevator, I found out that it needed 9.6 seconds to travel, so I entered 10 seconds.



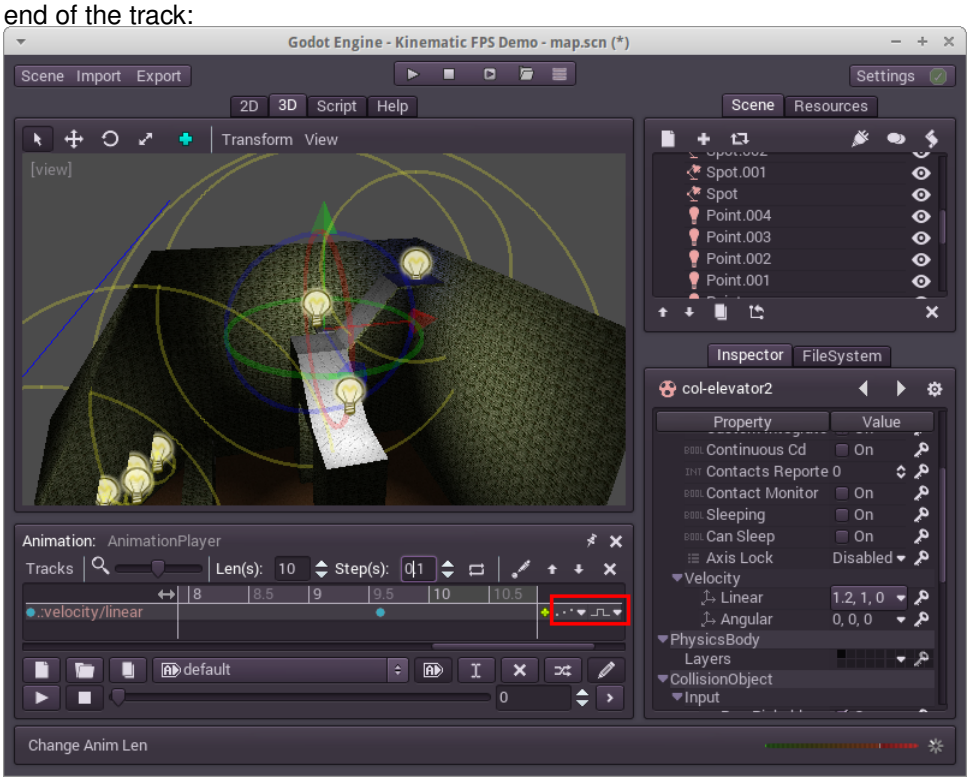
179) In the timeline, go to your desired end time, for me 9.6[s], and create a *Linear Velocity* for the value [0,0,0]. That will stop the elevator.



Try it and see how goes your elevator. Adjust the timing and velocity until it seems good.

Hint

I almost forgot. The velocity must be constant during the whole travel. And for that, the track of the *Linear Velocity* that you are creating must be of type *Discrete* and *Nearest*. You can change that at the very right end of the track:

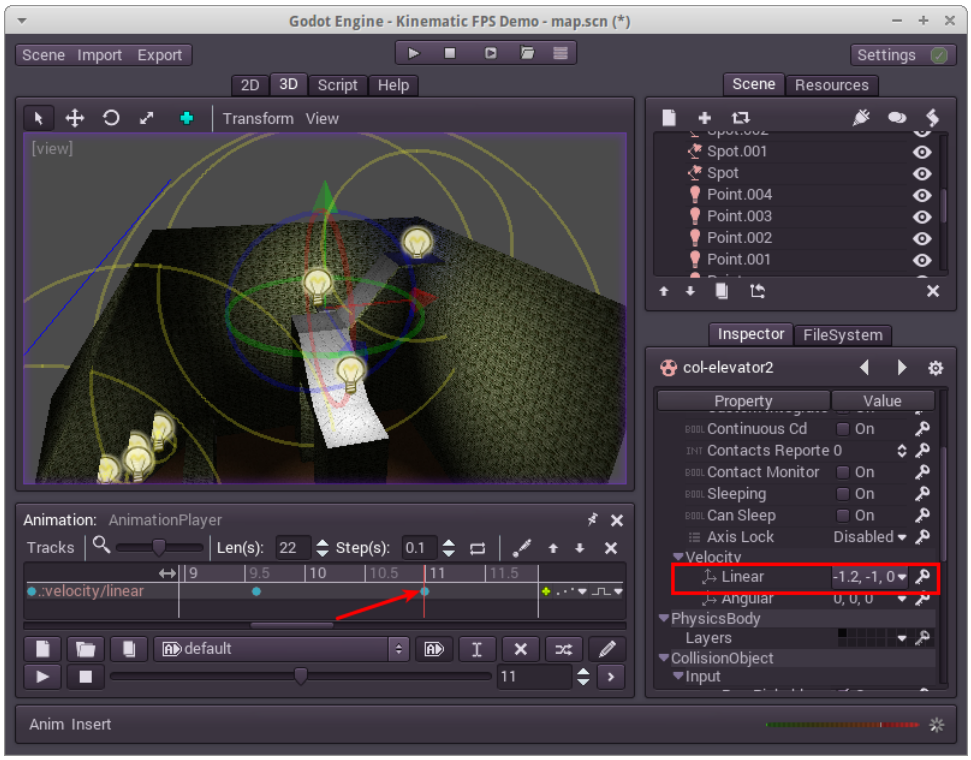


Now, we must make the elevator go back to it's original point, after a little pause.

180) Change the length of the animation to the double plus the pause time. Since my animation was of 10[s], I set 22[s].

181) Go in the timeline a little after the end point of the animation, where you created a *Linear Velocity* key with value [0,0,0]. I choose 11[s] to give a second and half of wait.

182) Create a key with the opposite value of the *Linear Velocity*. For me, that gives [-1.2,-1,0].



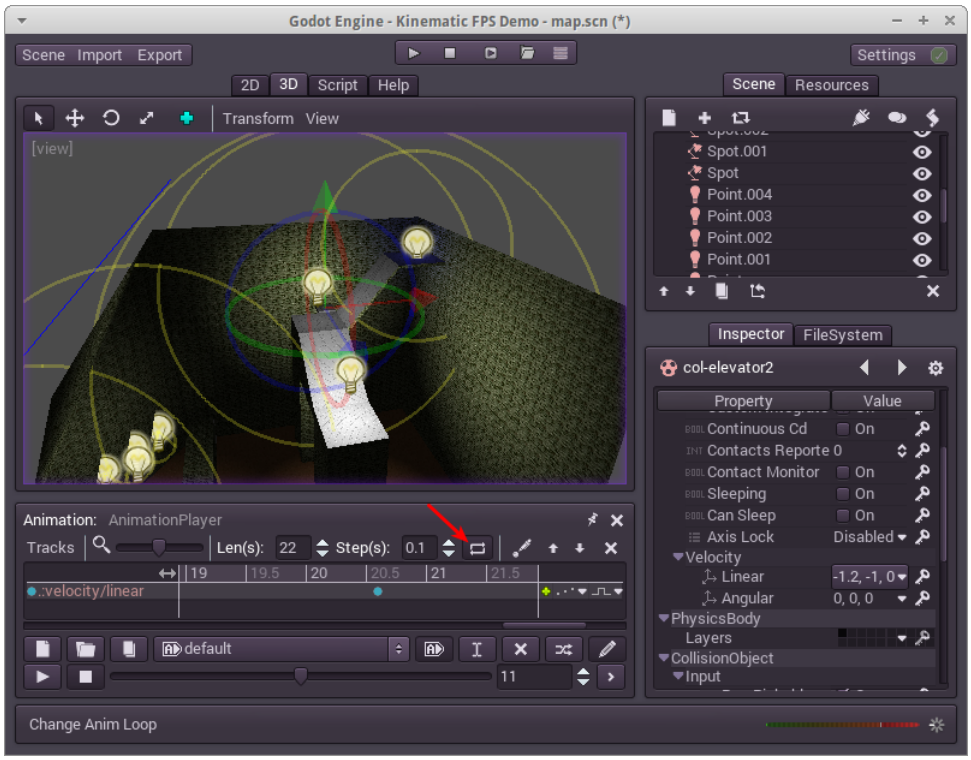
183) then go at the time where the travel back animation should end, and create another key with value [0,0,0]. I choose time 20.5[s] because the starting time + travel time are 11 + 9.6 [s]

Important!

The timing must be perfectly calculated. If the time and velocity to travel back are not the same as the ones in the first travel, the elevator won't come back exactly to where it was originally. And since this animation will be played in a loop, any difference will grow bigger at each loop. Eventually the elevator completely stray from its original path.

Erratum This is not needed anymore if you animate with *Translation* instead of linear velocity.

184) And to conclude, click on the loop button.



You can then close the animation editor and try it.

Funny! If you leave it alone, it does travel forth and back properly. But if you try to step on it and stay on it, it moves differently. It has trouble to go up. Your actor interferes with the elevator path.
But don't worry. This can be greatly minimized with scripting.

Erratum With the transpose animation, the elevator sticks to its path. However the actor has trouble staying on the elevator. This will be fixed too in the next chapter.

8.3 Implementation of the floor velocity

Now we just have to add a little code in our actor to complete the implementation of elevators.

185) Go back to the Script window and back to the "actor.gd" file.

186) Append at the end of the file:

```
func _get_floor_velocity(ray,delta):
  var floor_velocity=Vector3()
  # only static or rigid bodies are considered as floor. If the
  # character is on top of another character, he can be ignored.
```

```
  var object = ray.get_collider()
  if object extends RigidBody or object extends StaticBody:
    var point = ray.get_collision_point() - object.get_translation()
    var floor_angular_vel = Vector3()
    # get the floor velocity and rotation depending on the
    # kind of floor
    if object extends RigidBody:
      floor_velocity = object.get_linear_velocity()
      floor_angular_vel = object.get_angular_velocity()
    elif object extends StaticBody:
      floor_velocity = object.get_constant_linear_velocity()
      floor_angular_vel = object.get_constant_angular_velocity()
    # if there's an angular velocity, the floor velocity take it
    # in account too.
    if(floor_angular_vel.length()>0):
      var transform = Matrix3(Vector3(1, 0, 0), floor_angular_vel.x)
      transform = transform.rotated(Vector3(0, 1, 0), floor_angular_vel.y)
      transform = transform.rotated(Vector3(0, 0, 1), floor_angular_vel.z)
      floor_velocity += transform.xform_inv(point) - point

      # if the floor has an angular velocity (rotation force),
      # the character must rotate too.
      yaw = fmod(yaw + rad2deg(floor_angular_vel.y) * delta, 360)
      get_node("yaw").set_rotation(Vector3(0, deg2rad(yaw), 0))
  return floor_velocity
```

Hint

This function calculates the linear velocity of the ground where the actor is standing on, by using the raycast *ray* to detect the collision and the detect object of collision. And then, with some voodoo matrix transformation (in fact, it's just rotating the linear velocity vector with the angular velocity), a floor velocity is calculated. And for convenience we rotate the *yaw* of the actor here, but only if there is an angular velocity.

187) We'll now call this function and make the actor move with the floor velocity. At the end of the *_walk* function, replace the code:

```
if on_floor:
  # jump
  if Input.is_action_pressed("jump"):
```

with:

```
if on_floor:
  # move with floor but don't change the velocity.
  var floor_velocity=_get_floor_velocity(ray,delta)
  if floor_velocity.length()!=0:
```

```
    move(floor_velocity*delta)

  # jump
  if Input.is_action_pressed("jump"):
```

Give it a try. Doesn't look bad, eh? Now the actor stays properly on the elevator and doesn't slip out of the elevator. And the actor doesn't interfere with the elevator course, except if you really try and jump on the elevator lot of times. And even so, it is hard to notice the difference (but yes, that's still a bug). That's one limitation of this way of doing.

Erratum If you followed the previous erratums, you don't have this limitation. By setting the body to kinematic and by moving the node with translation, the linear velocity is automatically updated.

You can now do the same for all others moving platforms.

Hint

You can find the complete script and scene for the actor at this point in the project's folder "tutorial/floor_velocity". The scene of the map is in the "source/game_assets" folder.

9 Ladders

Implementing ladders is actually really easy to do. Remember the fly mode? Climbing ladders is the same as flying, but only in a restricted area. Thanks to *Area* nodes, you can switch between fly and walk mode. And that's all we need to implement.

9.1 Adding Areas

This time, we need to have both the map and the player in the same scene.

188) Open "main.scn" scene.

189) Select the map node, probably still named *Spatial*.

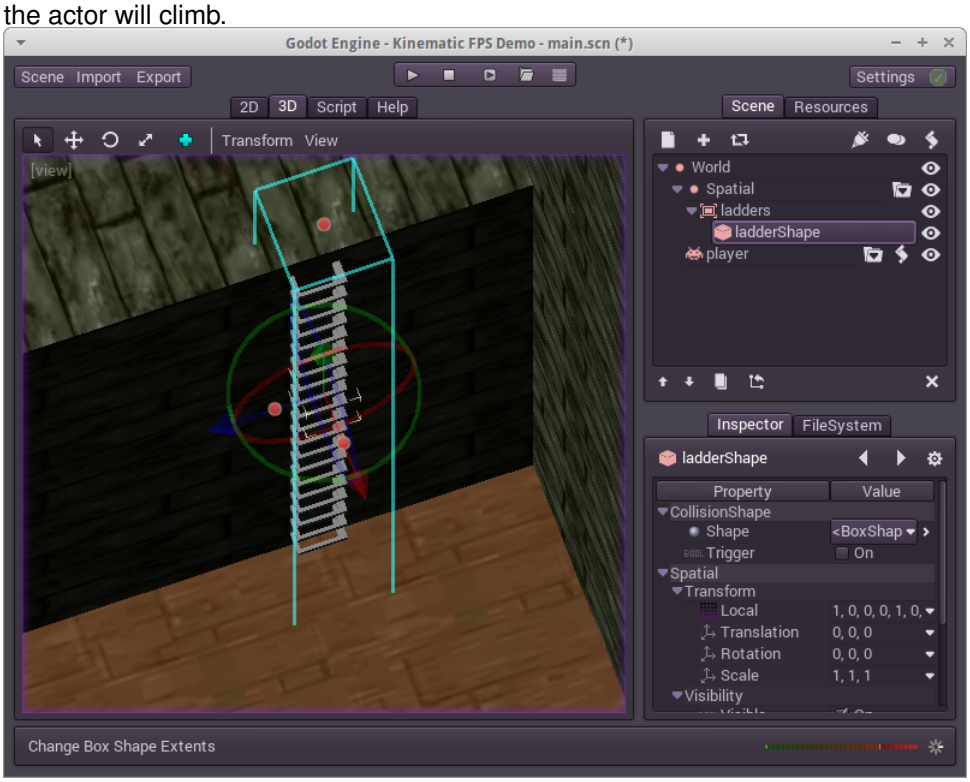
190) Add an *Area* node and move it to your ladder's position. Rename it with a better name, like *ladders*.

191) Add a *CollisionShape* to this node and rename it like its parent, like *ladderShape*.

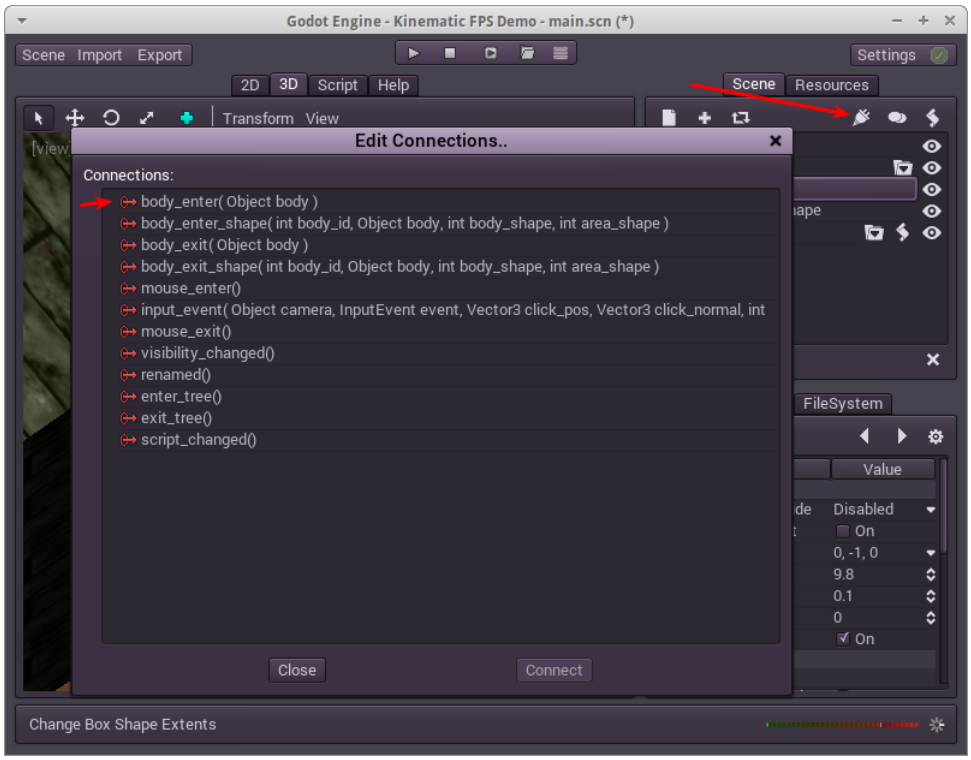
192) In this *CollisionShape*, create a *BoxShape*.

193) Resize this *BoxShape* to the size of the ladder and the space where

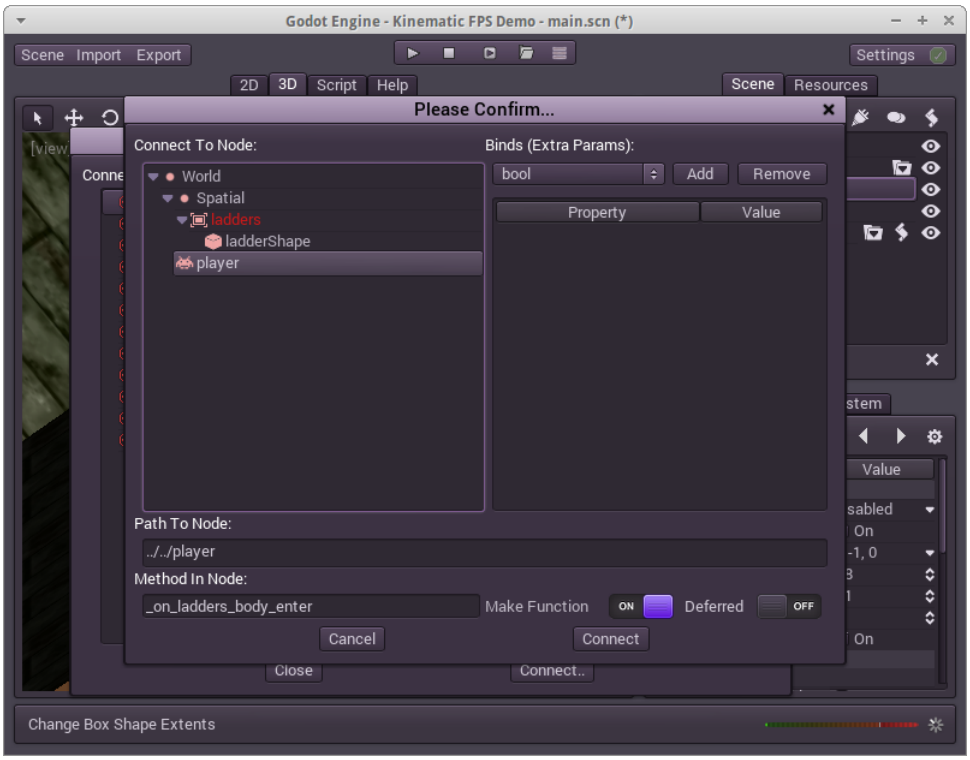
the actor will climb.



194) Click on the *Area* node, here *ladders*, and make a new node connection for *body_enter*:



195) Connect it with the *player* node, with default options.



196) repeat the last 2 steps for the *body_exit*
Now you should have two empty functions in the ”actor.gd” script.

9.2 Implement the fly/walk switch

197) In the script file, add in the fields section:

```
var fly_mode=false
```

198) Replace:

```
func _fixed_process(delta):
  _walk(delta)
```

with

```
func _fixed_process(delta):
  if fly_mode:
    _fly(delta)
  else:
    _walk(delta)
```

199) Replace:

```
func _on_ladders_body_enter( body ):
  pass # replace with function body

func _on_ladders_body_exit( body ):
  pass # replace with function body
```

with

```
func _on_ladders_body_enter( body ):
  fly_mode=true

func _on_ladders_body_exit( body ):
  fly_mode=false
```

Let's try it.

When you enter the area, you just have to look up or down and move forward to climb up or down. And if you leave the ladders in the middle of the way, you fall. Simple but efficient. And if you made it high enough above the ladders, you can easily climb down the ladders from the top.
If you have a hard time to climb up, that's probably because the area or the cube were misplaced and are too far in the wall. And if you were too generous with the area size, you climbs the ladder from too far. It only needs proper adjustments of the size of the box.
For every others ladders, you just have to connect them to the player with the existing functions (no need to create new ones).

In my demo, there was some strange bug when climbing up the ladders, where we move not fluently. But that's because the collision shape of the ladder is a bit too complex. To avoid that, keep the collision shape of the ladder as simple as possible, like a box. You can just use a ladder texture on the wall if you want. As long as the area node is properly placed, it will work.

Hint

You can find the complete script and scene for the actor at this point in the project's folder "tutorial/ladders".

10 Doors & buttons

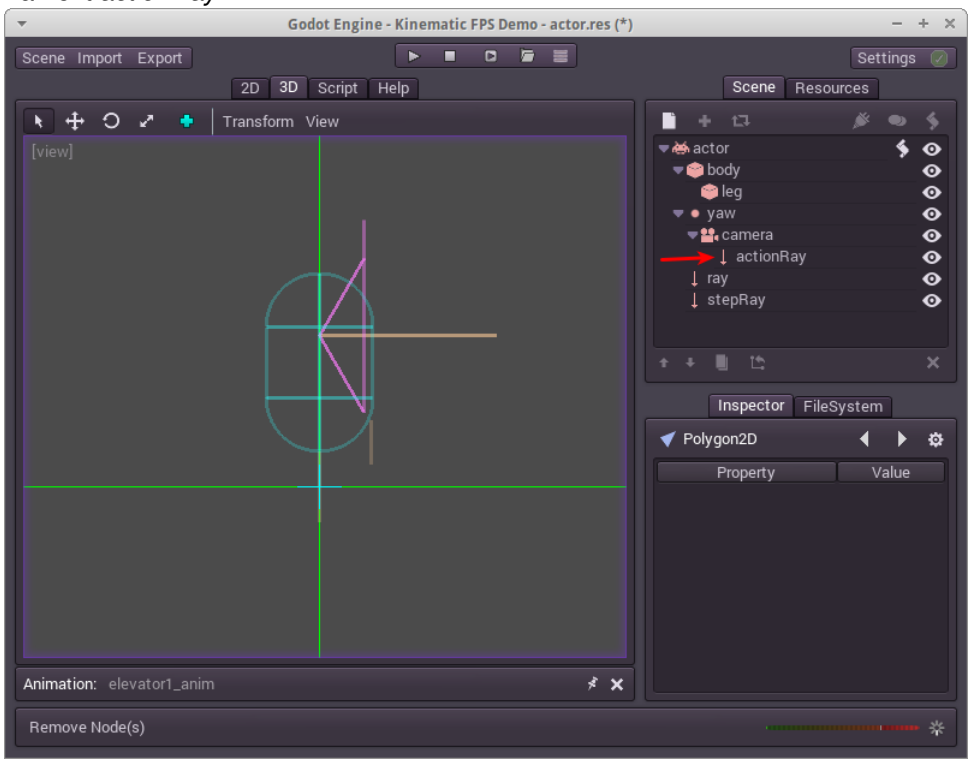
The last point we will cover are the interactive objects. They are objects that start an event when the player interact with them, for instance buttons.

10.1 Prepare the actor

The actor, especially the one controlled by the player, will have a raycast that represents the focus of the actor. Everything in focus can be used. And by that, I mean to call a function from the object's script, which would do whatever this object is supposed to do when activated. For instance, if the object is a button, when the player focuses on this button and presses the *action* key, the actor's script will call the *on_pressed* function of the button, which would for instance play an animation that makes the elevator move.

200) Open the actor scene.

201) In the scene tree, select the *camera* node and add a *RayCast*. Rename it *actionRay*.



202) In its properties, mark *Enabled* and set its *Cast To* to [0,0,-2].

203) Go in the script "actor.gd", and add in the *_ready* function:

```
get_node("yaw/camera/actionRay").add_exception(self)
```

Hint

The raycast stops at the very first object it encounters. In our case, the *actionRay* would always collide with the actor. So, we add an exception to not collide with the actor. This function is available only in Godot v1.1 and later. If you really want to use godot v1.0, you have to move the raycast outside the body.

204) In the *_input* function, append:

```
if ie.type == InputEvent.KEY:
  if Input.is_action_pressed("use"):
    var ray=get_node("yaw/camera/useRay")
    if ray.is_colliding():
      var obj=ray.get_collider()
      if obj extends Button_class:
        obj.on_pressed()
```

205) And in the fields section, add:

```
const Button_class=preload("res://button.gd")
```

206) In a separate text editor, like Notepad, create a file that contains:

```
extends PhysicsBody

func _ready():
  # Initalization here
  pass

func on_pressed():
  print("action!")
```

207) Save it in the ”source” folder with the filename ”button.gd”.

Hint

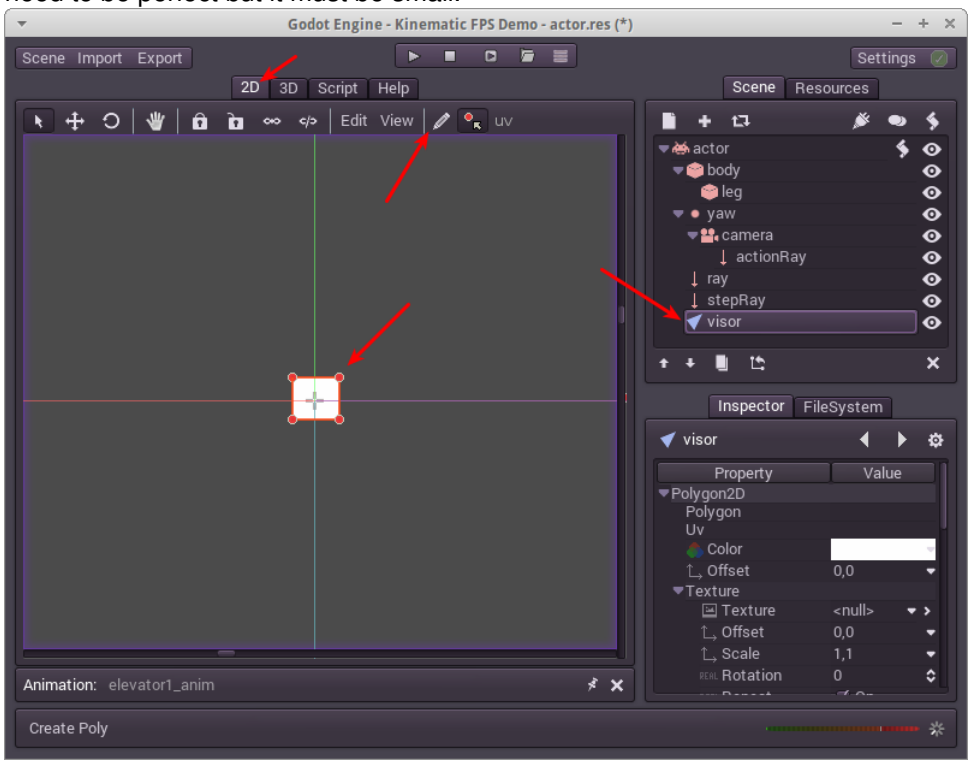
As you can guess, the word ”action!” will be called by the actor when he ”uses” the object that uses this script.

Let’s add a little visual help. The raycast being not visible, we need a visor to know where the raycast points to.

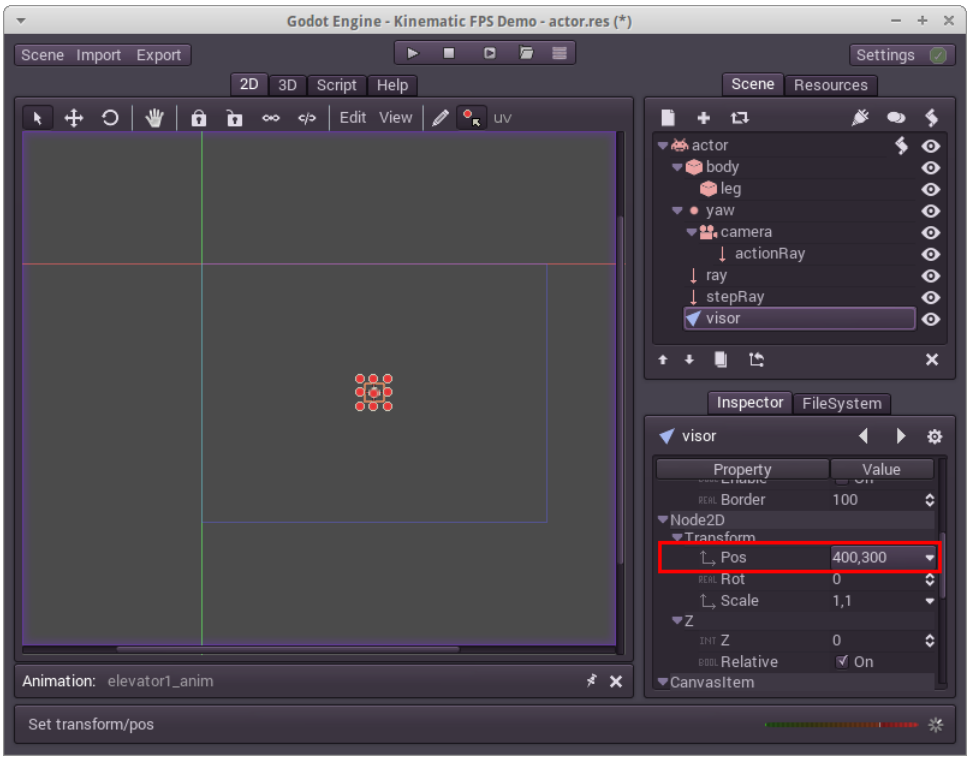
208) In the scene tree, select the *actor* node.

209) Add a *Polygon2D* node and rename it *visor*.

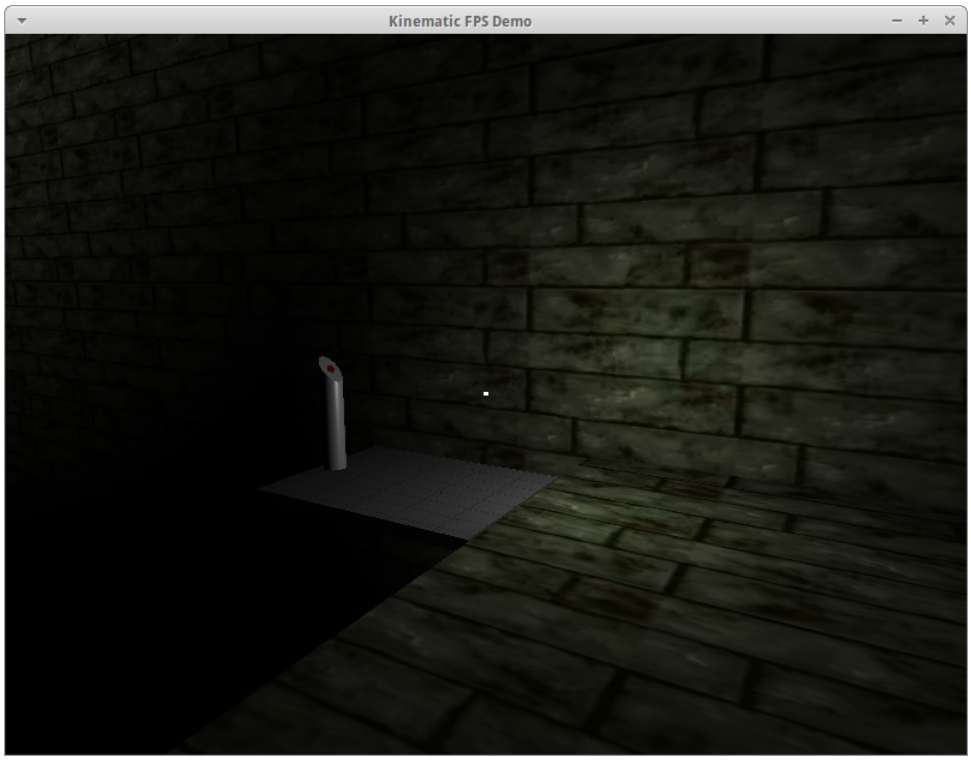
210) Zoom in a lot and draw with the pen tool a small rectangle. It doesn't need to be perfect but it must be small.



211) Move it to the center of the window. Since it's a resolution of 800x600 for my project, I set it to [400,300].



The actor is now ready. If you try it, you should now see a small white rectangle in the middle of the screen. That's where your focus is. But for now, you can't interact with anything.

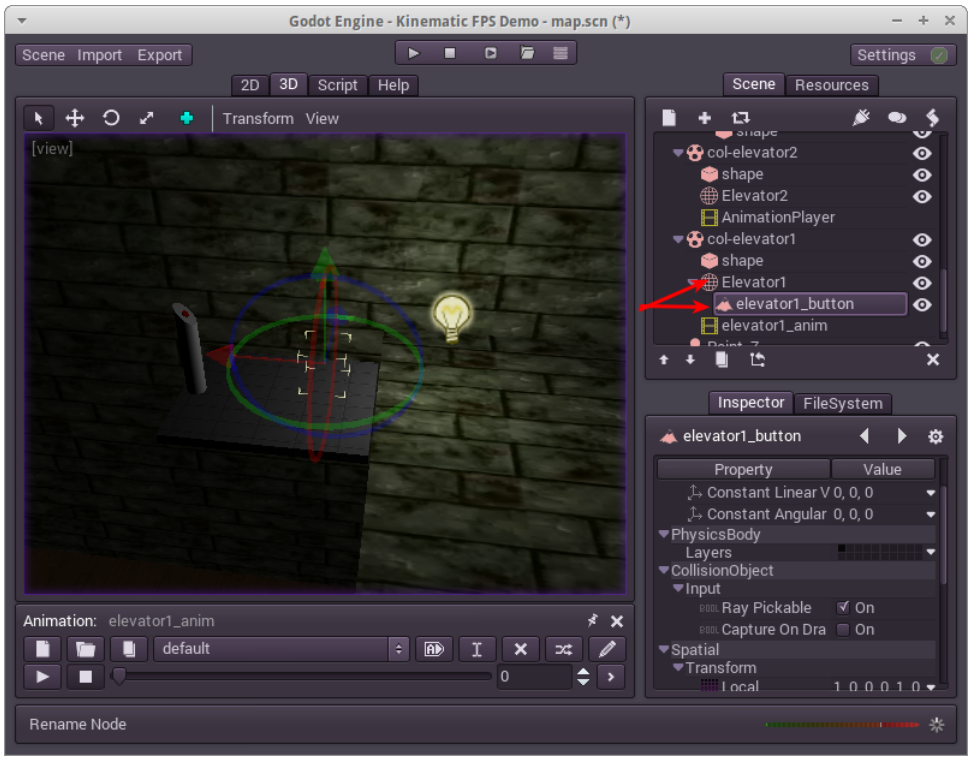


10.2 Prepare the button

We are here going to implement a button that makes an elevator move. I'll suppose that you have an elevator with an animation ready. Just make sure that the animation of this elevator is not in autoplay mode. We want the elevator to stand still by default.

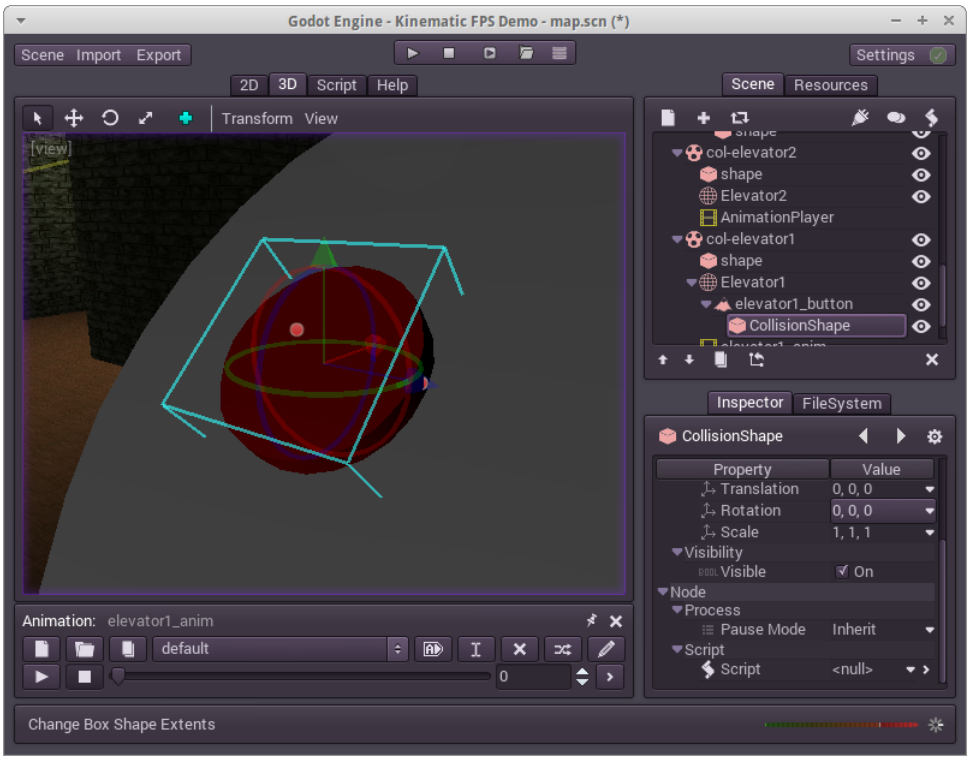
212) Open the map scene.

213) Select the mesh node with the button. Add a *StaticBody* and rename it *elevator1_button*.



214) Add to this *elevator1_button* node a *CollisionShape*, add give it for shape a *BoxShape*.

215) Now resize and move this static body where the button mesh is.

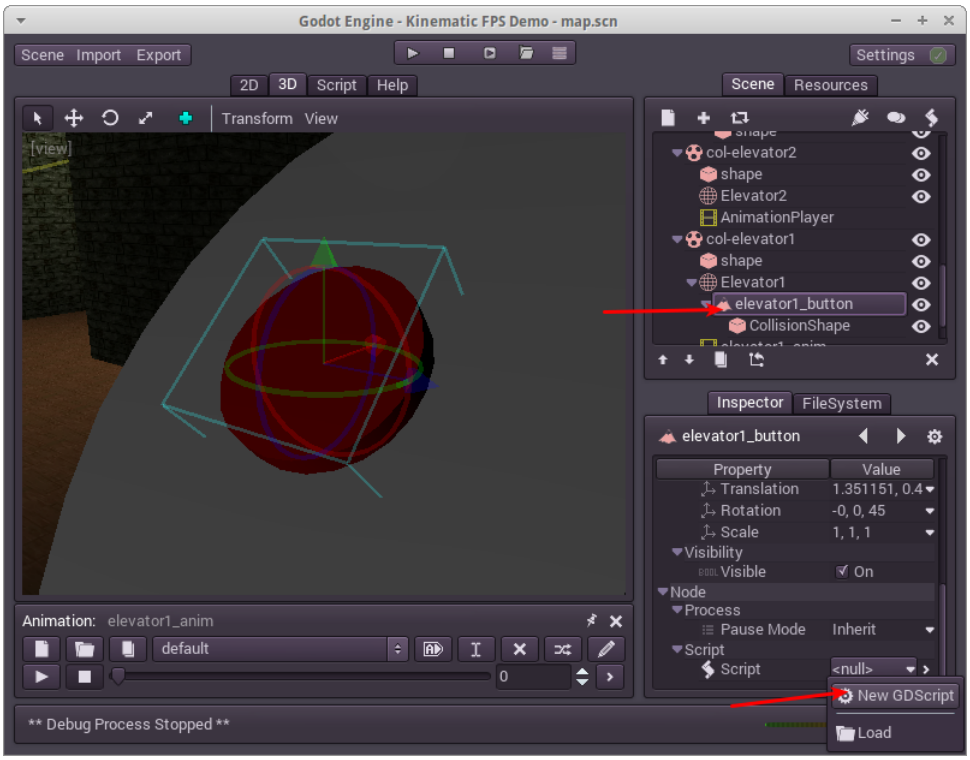


216) In the *CollisionShape* node you've just created, mark the property *Trigger* as On.

Hint

This property disables the collision with the elevator, which would otherwise be pushed and start to move, and this static body will only serves to be detected by the actor's raycast.

217) Select the *elevator1_button* node and create a new script.



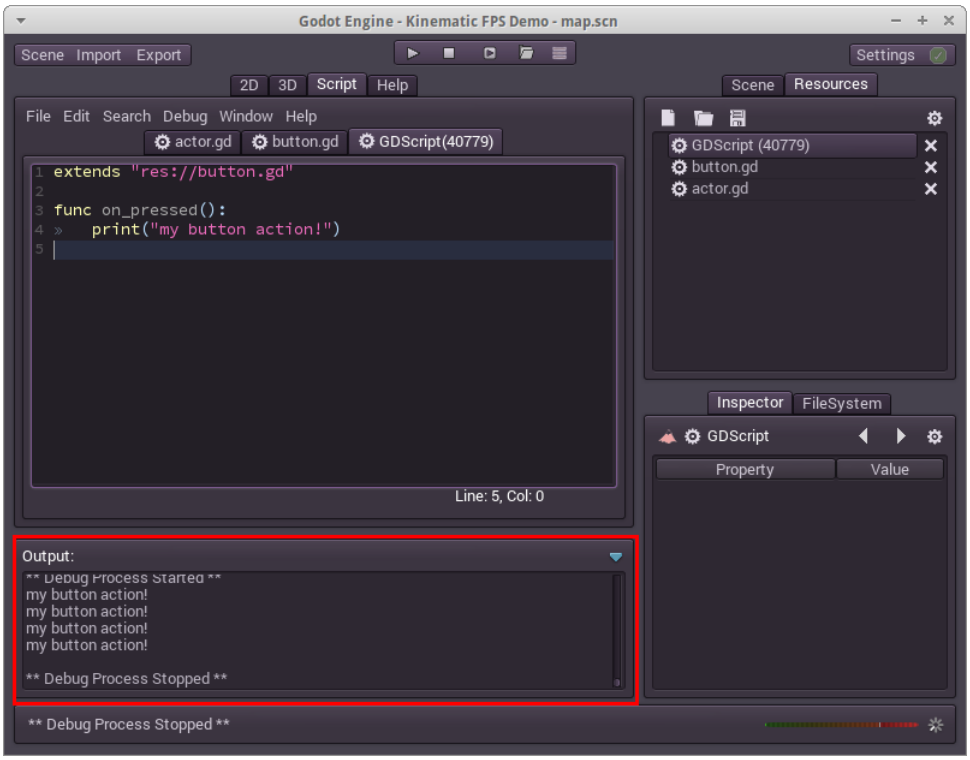
218) Enter in this script.

219) Add the following code:

```
extends "res://button.gd"

func on_pressed():
  print("my button action!")
```

Try it and watch out the console. Now go to the button and press the ”use” key while putting the visor in the middle of the button. The console shows ”my button action!”.



Great, it works! ...Wait. What? But the ”button.gd” script should print ”action!”. Instead it prints the text of the embedded script of the button node. That's simply because the embedded script overrides the button class it's extending. If you remove the *on_pressed* function in the embedded script, it would print the ”action!” text. This mechanism is the class inheritance. More information in the wiki [3]. It allows you to write a custom behaviour per node, and the node will still be considered as a button.

Now the last touch, make the elevator move.

220) Replace the print call with:

```
var anim=get_node("../../elevator1_anim")
if !anim.is_playing():
  anim.play("default")
```

Maybe you have to adjust the *get_node* path and the animation name. For me, the animation player was two parent nodes higher and with the name *elevator1_anim*. And I called my animation *default*.

Try it. It works! The elevator moves, and only when we press the button. We finally implemented a working button. Let's do the same for the door, shall we. But this time I let you do it yourself.

221) Select the door node and give it a simple open/close animation with

an animation player.

222) Like for the button, create an embedded script for the static body of the door (by default *col*) and a code similar to the button's one, with adequate node path to the animation player and animation name.

And this it it! With a simple code, you implemented a door exactly like the button. Why didn't we create an invisible new static body for the door? That wasn't needed since the whole door could be considered as a button. So we used the already existing static body of the door.

Hint

You can find the complete script and scene for the actor and the button at this point in the project's folder "buttons".

11 Conclusion

Congratulation! You finished this tutorial and you made a FPS game.

It's not completely perfect, that's right. And there aren't even enemies nor weapons. But I let you have the joy to implement them. You have now the basics of a first person view controller, as well as the way to handle buttons, doors and interactive objects. You even have moving platforms and elevators.

You can find the whole project with sources at github [1].

11.1 What next?

From now on, you can implement :

- NPC and enemies. It's basically the same as the actor controller, but without camera nor controls from the mouse and the keyboard.
- Bullets. Otherwise it would hardly be called First Person **Shooter**.
- Items to interact with, or to take when you walk on them.
- Better door animation, with collision detection if an actor is in the way.
- Crouch and sprint.
- Sound effects and music.
- And all the others stuff like an HUD, load levels, menu...

I hope you had as much fun doing this tutorial as much as I had when writing it, and I wish you fun for your game project.

See you at the next tutorial!

12 Annexes

12.1 Camera aiming

In chapter 7.2, we used the following code to get the camera's rotation:

```
var aim = get_node("yaw/camera").get_global_transform().basis
```

In a nutshell, it's the rotation of the camera as a transform matrix (more explanation about transformations and quaternion in the wiki page of Godot [3]). This matrix contains both the horizontal rotation, the *yaw*, and the vertical rotation, the *pitch*, because it's the global transform (if you called the *get_transform()*, you won't get the *yaw* since this rotation is in the parent node).

And since the spatial reference of the camera uses the Y axis for elevation and is directed to the Z axis, it means that the 3rd column of the matrix, which is the Z axis, can be used to move forward and backward simply by adding it to the velocity. And it the same for the first column of the matrix, the X axis, to slide left or right.
In figure 14, which is a view from the top of the camera (Y axis is then ignored), the main node of the actor has no rotation and has for axis the global axis of the scene. The camera is looking in the direction of the aim (in blue), and that means that its local X and Z axis (in red) are rotated in the direction of aim (Y too, but we don't see it here). Since what is moving is the main node of the actor and not the camera, to move forward from the camera's point of view means that we just need to get the local Z vector and add it to the main node's position. And the good thing is that this local Z vector can have a Y value if the camera is looking a bit above or below. That means that in the fly mode, moving forward means not only to move in the horizontal XZ plane but vertically too.
This is the big advantage of the math with vectors instead of math with euclidean angles.

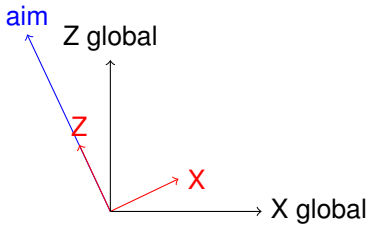


Figure 14: Top view of camera's aiming and transform matrix

12.2 Projection on a plane based on a normal

In the chapter 7.3.4, we talk about the velocity that should follow the ground. In chapter 7.3.5, we implement this concept with a mere line of code. In this chapter, I'll explain in a mathematical aspect how we calculate the new velocity.

Dot product But first, a little reminder. The dot product of a vector with another vector is the scalar projection (figure 15) of the first vector on the other one. One property of the scalar projection is that the vector to project (here a) is the hypotenuse of a right triangle with c as one orthogonal side. And c is oriented like b but with a different length.

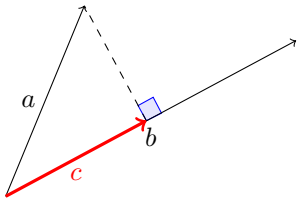


Figure 15: Scalar projection

In GDScript, there is a vector function called *dot* that returns the length of the projected vector. To get the actual vector like c in the figure, we must multiply the normalized vector b with the dot value, like:

```
var a=Vector3(...)
var b=Vector3(...)
var c = b.dot(a) * b.normalized()
```

It is important to notice that the other orthogonal side of the right rectangle is what we're looking for. In figure 16, the vector we want is d because it's perpendicular to vector b.

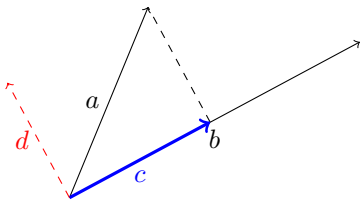


Figure 16: The vector d is perpendicular to vector b

To obtain vector d, we subtract the vector c to vector a, like in figure 17

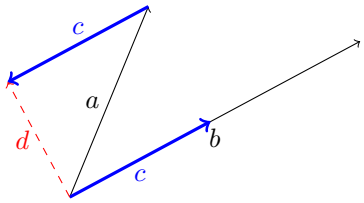


Figure 17: Subtraction of the scalar projection

In GDScript, the code would be:

```
var a=Vector3(...)
var b=Vector3(...)
var c = b.dot(a) * b.normalized()
var d = a - c
```

And after simplification, it's:

```
var a=Vector3(...)
var b=Vector3(...)
var d = a - b.dot(a) * b.normalized()
```

For our tutorial, a is the velocity and b is the normal of the collision, which is by default normalized. And d is the resulting velocity we want to apply to the actor. The final code as implemented is like:

```
velocity = velocity - normal.dot(velocity) * normal
```

You might wonder, why could this work for us since those are 2D vectors and what we use are 3D vectors? What about the 3rd dimension? In fact, it would make no difference if it was 2,3 or 80 dimensions. By using only 2 vectors of our game space, we worked in a non-orthographic 2 dimensions space made from those 2 vectors.

References

[1] Kinematic FPS demo project
https://github.com/gokudomatic/godot

[2] Godot Game Engine *Okam Studio* 2014.
http://www.godotengine.org/wp/

[3] Wiki of the Godot Engine
Main wiki page : https://github.com/okamstudio/godot/wiki

Wiki page about Vectors : https://github.com/okamstudio/godot/wiki/tutorial_vector_math
Wiki page about quaternions : https://github.com/okamstudio/godot/wiki/tutorial_transforms

[4] FPS Test demo
https://github.com/leezh/godot

[5] Blender
http://www.blender.org/

[6] Sketchup
http://www.sketchup.com/

[7] Sketchup STL export script
http://www.guitar-list.com/download-software/convert-sketchup-skp-files-dxf-or-stl

[8] Fox2d Demo
https://github.com/gokudomatic/godot

[9] Doom
http://www.idsoftware.com/

[10] Freedoom project
https://freedoom.github.io/