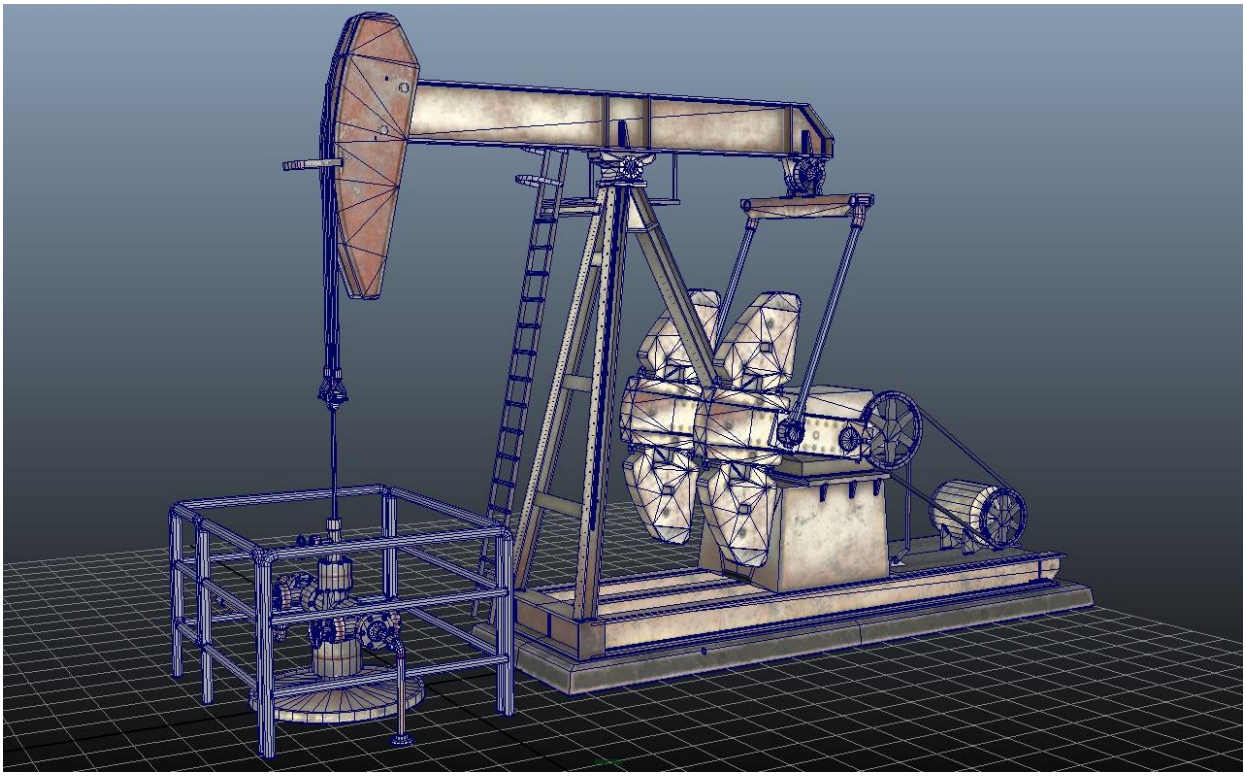
Custom Level Entities: Models, Overlays, and PbrMaterials

Project: **SnowRunner™**

Version 0.9.4 of Oct 12, 2022



Contents

Revision History

Version	Changes
0.9.2	Initial public version.
0.9.3	• Changed the document name (“Creating a Custom Model” -> “Custom Level Entities. Models, Overlays”). • Changed the document structure. Added the information about custom overlays creation, and changes in other sections on tags.
0.9.4	• Changed the document name (“Custom Level Entities. Models, Overlays” -> “Custom Level Entities: Models, Overlays, and PbrMaterials”). • Changed the document structure. Added the information about custom PbrMaterials: 4. Custom PbrMaterials, 8. Tags and Attributes of PbrMaterials

1. Introduction

This guide describes the creation of custom models, overlays and PbrMaterials for SnowRunner™ Editor. After their creation and integration with the Editor, these models, overlays and PbrMaterials can be used for creating new maps along with the regular models, overlays and PbrMaterials.

2. Custom Models

This section provides information about how to create a custom model and use it in the Editor.

2.1. Files of the Model

Usage of a custom model in the Editor requires the following files in the following folders:

Folder	***Files***	***Comments***
Media\meshes\models	**.fbx** file of the mesh **.xml** file of the mesh	The **.fbx** file of the model contains its mesh and its collision mesh. The **.xml** file of the mesh, stored in the same folder, contains data on materials (textures) of the model.
Media\textures\models	**.tga** files of textures	This folder should contain all textures that were assigned to the mesh of the model in the **.xml** file of the mesh.
Media\classes\models	**.xml** file of the class	The **.xml** file of the class of the model defines its properties: instancing, physical model, collisions, etc.
Optional (if you want to create a landmark model for your model):		
Media\meshes\landmarks	**.fbx** file of the landmark model **.xml** file of the mesh of the landmark model	If you want your model to appear on the navigation map, the **.fbx** file and **.xml** file of the mesh need to be created for the *landmark model*. See 2.2.7. Optional: Landmark model for details.

The **meshes\models**, **textures\models**, **classes\models**, and, optionally, **meshes\landmarks** folders must be created in the **Media** folder, which is located in the **Documents\My Games\SnowRunner** folder. The full path to the **Media** folder is typically similar to the following:
C:\Users\<name_of_user>\Documents\My Games\SnowRunner\Media

NOTE: If you are using the Public Test Server (PTS) version of the game, the path will be similar, but with **SnowRunnerBeta** instead of **SnowRunner**.

2.1.1. Important Note on File Naming

All names of created files can contain only small Latin letters, digits, and "_".

Moreover, there is another important convention rule:

You should use the same name for the following files (except the file extension):

- FBX file (in the **Media\meshes\models** folder).
  For example, “**oil_pump_01.fbx**”.
- XML file of the mesh (in the **Media\meshes\models** folder).
  For example, “**oil_pump_01.xml**”.
- XML file of the class of the model (in the **Media\classes\models** folder).
  For example, “**oil_pump_01.xml**”.

In this case, the system will be able to link all these files.

For trucks, we were required to specify the path to the XML file of the mesh in the mandatory **Mesh** attribute of the **<PhysicsModel>** tag in the XML file of the class.
As opposed to trucks, for models it is not necessary and we omit the **Mesh** attribute in the **<PhysicsModel>** tag. We can do it since the naming scheme allows the system to find the XML file of the mesh by the name of the XML file of the class.

2.2. Overview of the Process

The subsections below give a brief overview of the process of creating a custom model.

NOTE: This overview does not include the important rules that should be followed when specifying the properties of the model and particular tags used in XML files. They are covered in the next chapters.

The sample files used for this overview can be found in the 9.1. Sample Model section.

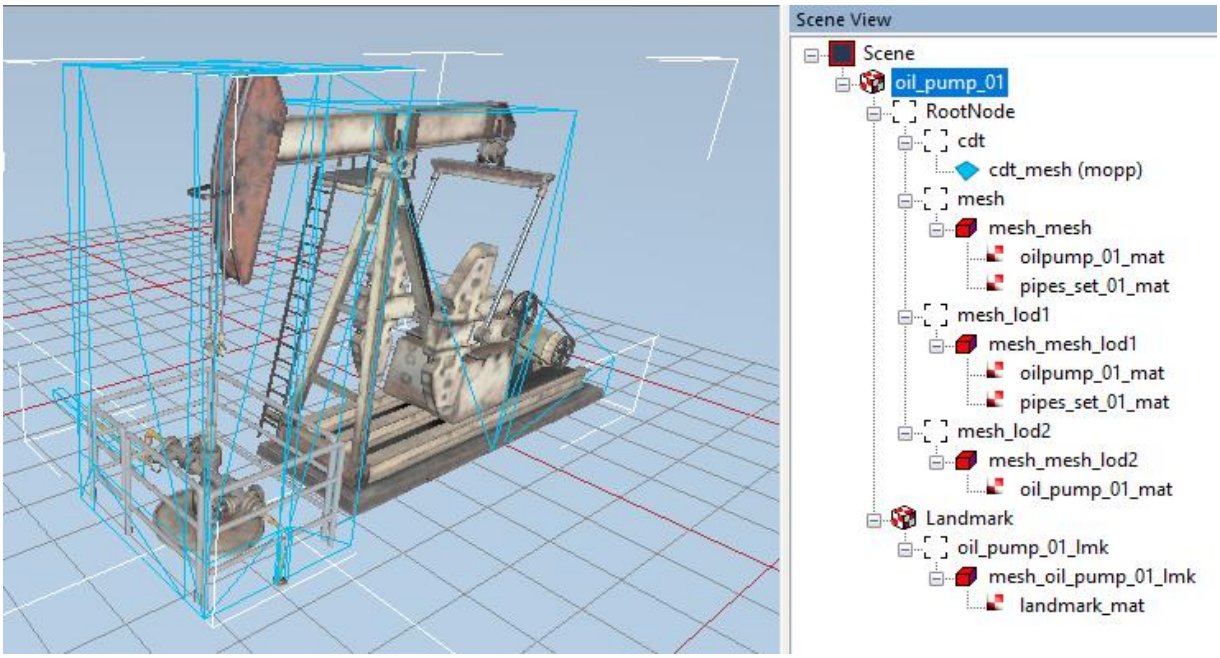
2.2.1. Create the model in the 3D Package

During the creation of the 3D model, you need to adhere to the following rules in the 3D modeling software you prefer:

- All meshes of the model must form a hierarchy, with one root node.
- The model must contain at least one visible mesh. Names of visible meshes must contain only small Latin letters, digits, and "_"
- The model must contain at least one collision mesh. Names of collision meshes should start with the "**cdt**_" prefix.
- The fewer number of visual meshes a model contains - the better (in terms of performance). In most cases and for the setup of regular models, it's better to use a single visual mesh.
- For dynamic and destructible models, we recommend you to use cdt meshes with primitive shapes (boxes, cylinders, etc.) that contain as few triangles as possible. When a cdt mesh has to be complex, it will be much better if it will be convex at least (for the lesser drop of performance).
- When creating visual and cdt meshes of the model, ensure that these meshes have exactly the same locations of pivots and exactly the same local transforms. Thus, we recommend you to use "freeze transforms" + "reset transforms" operations for these meshes in your 3D modeling software (names of these operations may vary depending on the particular modeling tool). If these pivot locations and transforms are different, this may result in various issues (e.g. rotation of the model in the Editor will not match the model rotation in the game).
- All objects and meshes ***must have the identity scale***. For example:
    - in Maya - it corresponds to the (1, 1, 1) scale,
    - in 3DS Max - to the (100, 100, 100) scale.
    - in Blender - to the (1, 1, 1) scale.

  Ensure that the root frame and all meshes of the model have the identity rotation (0, 0, 0).

Sample hierarchy of meshes of a custom model displayed in SnowRunner Editor is shown below:



2.2.2. Export a model to the FBX format

SnowRunner uses the FBX file format for 3D models. So, you need to export your 3D model to this format.
The operations here are similar to exporting your mod of the truck to FBX. For details, see the "**Exporting to Fbx: 3ds Max, Maya, and Blender**" guide, available at https://snowrunner.mod.io/guides/exporting-to-fbx-3ds-max-maya-and-blender.

After export, you need to put the resulting FBX file into the **Media\meshes\models** folder.

The name of the FBX file should contain only Latin characters, digits, and underscores ("_"). It must contain no spaces and no special characters (except "_"). Names with all lowercase letters are much preferable, though the upper case is not forbidden.
E.g. "**oil_pump_01.fbx**"

2.2.3. Create necessary textures for a model

You need to create textures for your model and assign them (as materials) to your 3D model during its creation in the 3D package.

After export to the FBX file, information about used textures is stored in the FBX file. However, for each texture, the system takes only its name from the FBX file. All properties of a texture are described in the XML file of the mesh (see 2.2.4 below).

NOTE: The fewer textures are assigned to a model, the better will be the performance of the maps with this model.

You may use the same texture maps on different parts of the model. But, for example, if one part should be able to become snowy in the game and the other one should be left unchanged, then you should take this into account at the stage of creating the 3D model. Particularly, in this case, you should create two materials with different names in the FBX file, since, in this case, you will be able to set different parameters for them in XML (the parameters responsible for snow cover). Each material in the FBX file corresponds to a separate set of parameters in the XML description.

You can use any textures for your model. However, they must conform with certain rules - see below.

Names of textures: Names can contain Latin characters, digits, and underscores ("_"). They must contain no special characters (except "_") and no spaces.

Suffixes in names: Textures should have corresponding suffixes ("*postfixes*") in their names, written after the original name of the texture. The typical texture set for a model should include a diffuse texture with an optional alpha channel, normal map, and shading map.

__postfix	***Type of Texture***	***Parameter in XML of the mesh (in the `<Material>` tag)***
__d	Diffuse (Albedo Map)	`AlbedoMap`
__d_a	Diffuse (Albedo Map) + Alpha	`AlbedoMap`
__n_d	Normal Map	`NormalMap`
__sh_d	Shading Map	`ShadingMap`
__em_d	Emissive Map	`EmissiveMap`

Format: The files of textures must be in the **.tga** format.

Texture sizes: For proper mipmapping and filtering, dimensions of a texture should be $\mathbf{2^N x 2^M}$. I.e., each side of the texture should be the power of two, and its width does ***not*** need to be equal to its height. Typical texture size for models should not exceed **2048x2048**. Higher texture sizes for models might impact the performance or texture streaming in general.

Location: You should put textures to the **Media\textures\models** folder.

2.2.4. Create the XML file of the Mesh

Now, you should create the XML file of the mesh for your FBX file in the **Media\meshes\models** folder. You should name the created XML file as the FBX file, e.g. "**oil_pump_01.xml**".

In this file, you need to specify the links to all textures used by the model and some other properties (see below).

Sample XML file of the mesh of the model:

```
<_templates Include="models" />
<CombineXMesh Type="Model">
	<MeshLod Distances="45, 140" HidingDistance="550"
MeshName="mesh" />
	<Material
		_template="DefaultNarrow"
		AlbedoMap="models/oilpump_01__d.tga"
		Name="oilpump_01_mat"
		NormalMap="models/oilpump_01__n_d.tga"
		ShadingMap="models/oilpump_01__sh_d.tga"
	/>
	<Material
		AlbedoMap="models/pipes_set_01__d.tga"
		Name="pipes_set_01_mat"
		NormalMap="models/pipes_set_01__n_d.tga"
		ShadingMap="models/pipes_set_01__sh_d.tga"
	/>
	<Material
		AlbedoMap="proxy/oil_pump_01__d.tga"
		Name="oil_pump_01_mat"
	/>
</CombineXMesh>
```

For details on tags and attributes used here, please refer to the 6. Tags and Attributes of Models chapter below.

As you can see, in the XML code here, we can inherit some code from templates. For more details on this feature, see the "***5.6. Templates (_templates)***" section of the **Integration of Trucks and Addons** guide (available at https://snowrunner.mod.io/guides/integration-of-trucks-and-addons-part-2). However, here we cannot create custom templates and can only use the existing ones, stored in the **initial.pak** archive, in the **[media]_templates** folder. For example, here we specify that we will use templates from the **models.xml** file there. And then, use the **DefaultNarrow** template, which is contained by **models.xml**.

Assigning textures to the model is also described in the "***7.2. <Material>***" section of the **Integration of Trucks and Addons** guide (available at https://snowrunner.mod.io/guides/integration-of-trucks-and-addons-part-3). The same properties of the material can be used here.
For each material, in the **Name** attribute, you need to specify the exact name of the material from the FBX file.
When defining textures of this material (`AlbedoMap`, `NormalMap`, etc.) you need to specify paths to the model's textures that you put into the **Media\textures\models** folder (see above). The paths here should be specified relative to the **Media\textures** folder.

NOTE: In the sample above, one of the textures is stored in the **Media\textures\proxy** folder and the path to it is a bit different (`"proxy/oil_pump_01__d.tga"`). I.e., you can have multiple folders with textures in **Media\textures** folder, if necessary.

2.2.5. Create the XML file of the Class

Create the XML file of the class of the model in the **Media\classes\models** folder.
You should name the created XML file as the FBX file and exactly as XML-mesh file, e.g. "**oil_pump_01.xml**".

In the created file, specify the properties of the model.

For example:

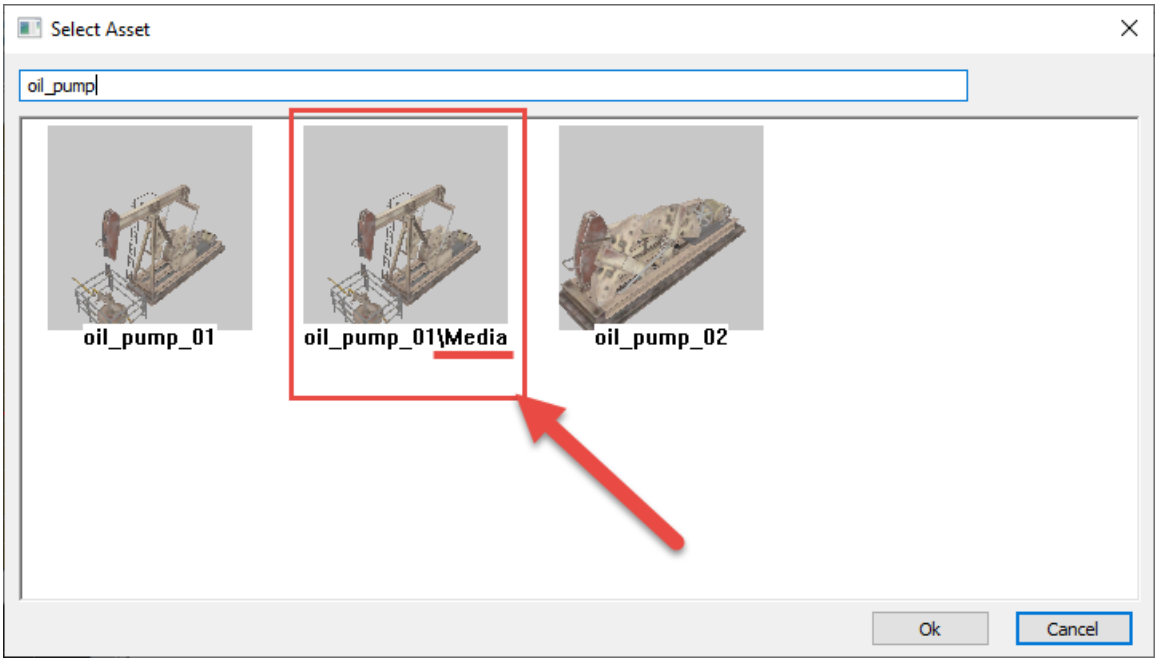
```
<ModelBrand
	Instanced="true"
	InstancedAsModel="true"
>
	<PhysicsModel>
		<Body Collisions="Dynamic" />
	</PhysicsModel>
</ModelBrand>
```

For details on tags and attributes used here, please refer to the 6. Tags and Attributes of Models chapter below.
NOTE: The XML-class file in the sample archive also contains the optional **<Landmark>** tag. For more details on it, see 2.2.7. Optional: Landmark model below.

2.2.6. Test the model in the Editor and the Game

If you have successfully created all these files, your custom model will appear in the Editor. You will be able to add it to the map the same way you add regular models.

In the **Select Asset** window, it will be displayed with the "**\Media**" suffix, to show that this is a custom model with source files in the **Media** folder.



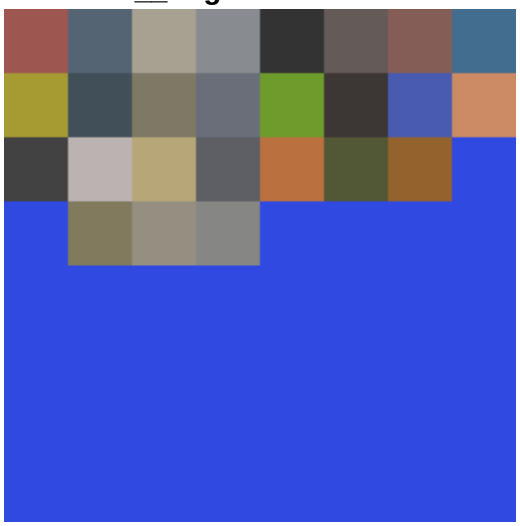
After adding the model to the map and packing this map, you can view your model in the game.

2.2.7. Optional: Landmark model

You may want to display some of your custom models on the navigation map. Typically, this is done for medium or large models (similar to a typical cargo unit or larger). For these models, you need to create and set up a landmark model.
To do this:

1. Create a landmark model in the 3D modeling tool of your choice.
   The created model must meet the following requirements:
   a. This must be a ***lowres*** model.
   b. This model is expected to be using only ***one simple texture with solid colors***. This is necessary to keep all models displayed on the navigation map in one style. The following texture should be used for all landmark models:

i. **landmark__d.tga**



ii. This texture is available in the **shared_textures.pak** archive. It is stored there as the **[textures]\pct\models_landmark__d.pct**

iii. In the XML-mesh file of the landmark model, you should specify the **"models/landmark__d.tga"** value as a path to this texture.

NOTE: This texture should be assigned to a 3D model using a single material, as a regular UV-mapping procedure for the mesh.

c. You need to use a **single material** for your landmark model. We strictly recommend that.

d. Naming: Typically, landmark models have the "**_lmk**" postfix in their names. E.g. **oil_pump_01_lmk**

2. Export this model to the FBX file the same way you did with the regular 3D model.
3. Create the **landmarks** directory in the **Media\meshes** folder (if this directory is not already created there). Put the FBX file of the model there.
   For example, in our case, the path to the FBX file will be **Media\meshes\landmarks\oil_pump_01_lmk.fbx**
4. Create an XML file of the mesh with the same name in the same directory.
   For example, in our case, the XML-mesh file will be **Media\meshes\landmarks\oil_pump_01_lmk.xml**
5. In this XML-mesh file, you need to assign our default texture for landmark objects (**"models/landmark__d.tga"**, see above) to the material from the FBX file of the landmark model.
   Thus, the contents of this file are very simple and typically look like the following:

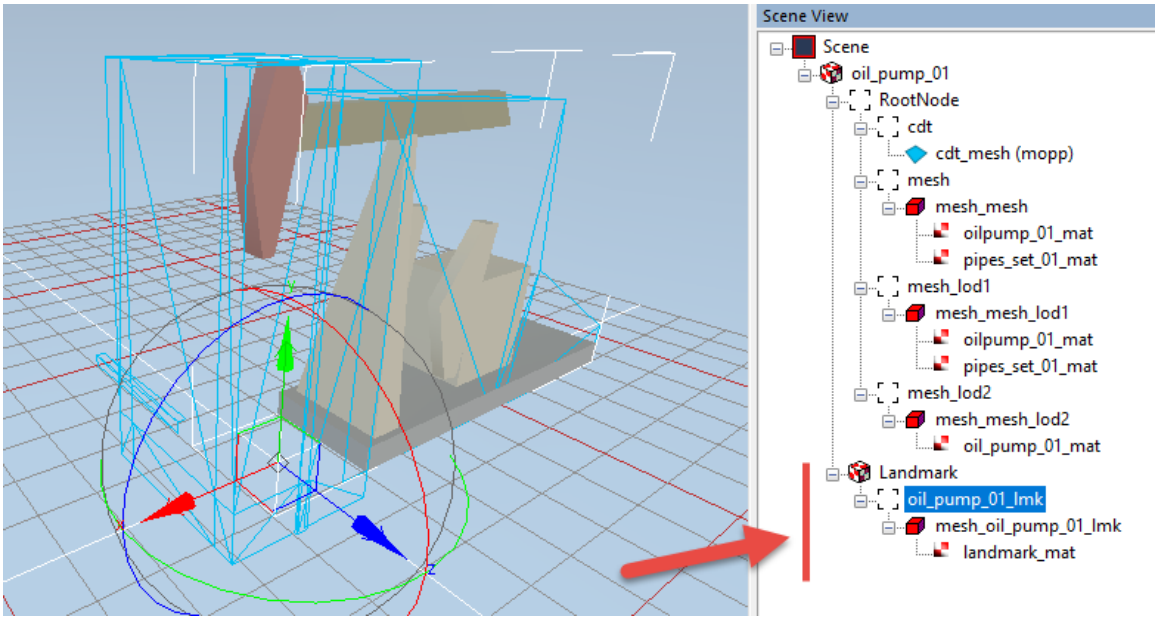
```
<CombineXMesh>
      <Material
            AlbedoMap="models/landmark__d.tga"
            Name="landmark_mat"
      />
</CombineXMesh>
```

where `landmark_mat` is the name of your material from the FBX file of the landmark model.

6. Now, you need to modify the XML class of the original model (not the landmark one, but the original one, e.g the "**oil_pump_01.xml**" in our example). You should add the <Landmark> tag there with the **Mesh** attribute linking to the XML-mesh file of the landmark model. This path should be specified relative to the **Media\meshes** folder and without the file extension.
   For example:

```
<Landmark Mesh="landmarks/oil_pump_01_lmk" />
```

7. If you open the XML-mesh file of the original model in the Editor, you will see that the **Landmark** section is displayed in the hierarchy of the model and it displays the contents of the FBX of the landmark model.



8. After repacking maps with this model, you will see that now your model is displayed on the navigation map.

2.3. Important Notes on Model Properties

2.3.0. Types of Models

There are 3 main types of models basing on their properties:

1. **Static** - these models do not move, have no animations, are not breakable.
   A model of this type contains no physical bodies with non-zero mass.
   For example: **initial.pak\[media]\classes\models\rock_03.xml**

2. **Dynamic** - these models can move and have collisions during this movement.
   A model of this type contains at least one physical body with non-zero mass. It can contain 1 constraint with Type="Fixed".
   For example: **initial.pak\[media]\classes\models\barrel_02.xml**

3. **Destructible** - these models are dynamic models that can be broken by a truck.
   A model of this type contains at least 2 physical bodies with non-zero mass and more than 1 constraint with BreakOffThreshold.
   For example: **initial.pak\[media]\classes\models\rus_fence_01.xml**

2.3.1. Rules for All Models

2.3.1.1. Rule #1: Attributes for Instanced Rendering

If your model is frequently used in the scene, you need to set up instanced rendering (instancing) for this model.
Instanced rendering should be set up according to the following rules:

1. ***For static models*** (these models do not move, have no animations, are not breakable), you need to set the following attributes in the `<ModelBrand>` tag to `true`:
   ```
   Instanced="true"
   InstancedAsModel="true"
   ```

2. ***For dynamic models*** (these models can move or are breakable), in the `<ModelBrand>` tag you need to set the `Instanced` attribute to `true` and you should ***not*** specify the `InstancedAsModel` attribute.
   I.e., you need to specify only the following:
   ```
   Instanced="true"
   ```

3. ***For models with animations***, instancing is not supported and we do not specify `Instanced` or `InstancedAsModel` attributes in the `<ModelBrand>` tag.

2.3.1.2. Rule #2: DynamicModel parameter

For dynamic models, you need to set `DynamicModel="true"`.

This is valid for all models that need the following:

- Ability to move between different terrain blocks correctly and have collisions during this movement. For example: barrels.
- Ability to switch between different visual and collision subsets (the `<Subset>` tag). For example: _**objective** models, that can be "built" during objectives (see "*5.15.3. Model Building Settings*" in "**SnowRunner™ Editor Guide"** available as the **"SnowRunner_Editor_Guide.pdf"** of the same documentation package).
- Ability to "sink" in the mud (partially).

NOTE: You should ***not*** enable the `DynamicModel` parameter when you use ***instancing for dynamic models***. I.e., `DynamicModel="true"` cannot be used simultaneously with `Instanced="true"`, if there is no `InstancedAsModel="true"`. For details on instancing, see 2.3.1.1. Rule #1: Attributes for Instanced Rendering.

WARNING: You should use the `DynamicModel="true"` option only for those models that do require it since enabling this option affects the performance.

2.3.1.3. Rule #3: Scenarios for Setup of Instancing for Various Types of Models

1. A model that is rarely used on the map (e.g. the unique building) - ***no instancing*** is required.

2. A model with animation sequences (e.g. windmill) - you should ***not*** use instancing for such models, since animations are not supported for models with enabled instancing.

3. A model with different visual and collision subsets corresponding to different states of the model (e.g. the _**objective** models, that can be "built" during objectives) - you should ***not*** use instancing for such models.

4. A static model that is frequently used on the map (e.g. a rock, a concrete block, a lamp pole, etc.) - we recommend you to use *instancing for static models*. I.e., we recommend you to set the following attributes in the `<ModelBrand>` tag to `true`:
   `Instanced="true"`
   `InstancedAsModel="true"`

5. A model with dynamic physics or physical constraints that is frequently used on the map (e.g. a barrel, a road sign, etc.) - we recommend you to use *instancing for dynamic models*. I.e., in the `<ModelBrand>` tag, you need to set the

`Instanced` attribute to `true` and you should ***not*** specify the `InstancedAsModel` attribute:
`Instanced="true"`

6. A breakable model that is frequently used on the map - *instancing for dynamic models* is recommended. I.e., in the `<ModelBrand>` tag, you need to set the `Instanced` attribute to `true` and you should ***not*** specify the `InstancedAsModel` attribute:
`Instanced="true"`

2.3.2. Rules for Configuring Physical Properties

From the point of view of game physics, most static models act like simple collision objects. They do not move, have no mass, they have no constraints specified for physical bodies within them. They can only experience collisions with other objects and may give friction to these objects in case of the contact. Moreover, for some specific static models collision can be disabled.

Dynamic models participate in the game physics more intensively. Along with experiencing collisions and friction, they can move, have a mass, have constraints specified for physical bodies within them, they can even be breakable.

2.3.2.1. Main Rules

1. Shapes of solid bodies must match the visual appearance of the model.

2. For most of the models (both static and dynamic ones), we need to set `Collisions="Dynamic"`. This means that the solid body of this model has collisions with other models with `Collisions="Dynamic"`. For example, this is valid for buildings that have barrels and other objects near them. Until no constraints are broken, these objects have no collision with the ground surface. When any constraint of a model becomes broken for the first time, the game engine adds the collision with the ground automatically. This mechanics is necessary to avoid breaking constraints in the moment of the activation of the body of the model (see `BreakOffThreshold` below). Moreover, the gravitation is also disabled for the model before the break of the first constraint.

3. For most dynamic models, we need to set `NoSoftContacts="true"`. This is necessary for hard contacts with trucks, to avoid the situation when wheels of the truck penetrate the model.

4. The camera can function in two modes for both static and dynamic models:
    - `ClipCamera="true"` - The camera is raised above the model. This is the default mode.

- `ClipCamera="false"` - The camera ignores the model and is able to move inside the geometry of the model. This mode is used for road signs, fences, poles, and so on.

5. For all dynamic models, we can use the mechanics of breakable constraints - `BreakOffThreshold`. The `BreakOffThreshold` attribute is specified for the `<Constraint>` tag (see 6.3.1.1.1. <Constraint>). This parameter defines the threshold for the force applied to the model before breaking the constraint of the model. I.e., if the applied force is greater than this threshold, this constraint of the model will become broken and will stop binding its bodies.
   Constraints with specified `BreakOffThreshold` are intensively used for constraints that bind the dynamic model “to the world” (to the point where the model is located on the map) and we recommend such usage.
   This approach is necessary to avoid situations when the model falls to the ground (if it is slightly above it) or when the model jumps from the ground (if it is placed within it). These effects can occur at the moment of the activation of the model when a truck drives near it but does not touch it.
   It is important to specify the appropriate value of the `BreakOffThreshold`. For example, if the constraint specified for a road sign will be too hard, then the truck will not be able to break it (because of that we recommend testing these constraints using the small vehicles). However, even hard constraints can break by themselves due to gravity (this depends on the shape of the model, its center of mass, and mass of the model). To avoid situations when these constraints break due to gravity on activation of the model, gravity for the model is disabled until the first of its constraints is broken.

6. The location of the center of mass of a dynamic model should be specified correctly. For most of the models, the center of mass is automatically located as needed - in the center of the shape of the model. This works for such objects as a cube or a barrel. However, for some objects, its default location is incorrect (e.g. for road signs). In this case, you can set its correct location by specifying the offset of the center of mass, e.g. `CenterOfMassOffset="(0; -1.6; 0)"`.
   If the location of the center of mass is incorrect and it is located outside the model or on the border of the model, the model can behave strangely in the game (e.g. as a roly-poly toy).

NOTE: For some details on the setup of physical properties for static and dynamic models, please refer to 6.3.1. <PhysicsModel>.

2.3.2.2. Rules for Particular Types of Models

NOTE: For main rules on configuring physics for models, see section 4.2.1.

For most models:
In the XML description of most models (both static and dynamic ones), it is convenient to use a template with `Collisions="Dynamic"`.
For dynamic models, this template should also include `NoSoftContacts="true"`.

You can find a lot of useful templates in the **initial.pak** archive, in the **[media]_templates\models.xml** file there.

NOTE: For info on templates, see the "***5.6. Templates (_templates)***" section of the **Integration of Trucks and Addons** guide available as the **"Integration_of_Trucks_and_Addons.pdf"** of the same documentation package).

Regular dynamic models: barrels, boats, and so on:

1. Ensure that the model is not initially placed below the horizontal surface or is "buried" in the snow or other objects. Otherwise, when the truck drives near it, the model will become "activated", its collisions with the surface/snow/objects underneath will be enabled, and, as a result, the model may start to behave strangely. For example, the piece of the fence which is buried in deep snow may try to jump up due to enabled collisions.
2. Along with that, you need to check the settings of the constraints and other things from the 4.2.1 section above.

Fences:

1. To eliminate collisions of adjacent sections of fences with each other, you need to set `Fence="true"` for all models of fences.
2. For most of the fences, `ClipCamera="false"` is necessary.
3. You need to check the location of the center of mass. Sometimes, it is located on the border of the model, which results in the "roly-poly toy" effect.
4. Constraints must be breakable. Avoid non-breakable constraints when the truck may get stuck while driving over the fence.
5. You can improvise and play with the hierarchy and different types of constraints to achieve a realistic effect of breaking the fence.

Road signs:

1. For most of the road signs, you need to set `ClipCamera="false"`.
2. You need to check the center of mass of a road sign. It should be located near the base of the road sign.

Bridges:
For bridges, you need to set the specific setting: `ClipCameraBridge="true"`.
In this case, the game engine will not raise the camera upwards (above the bridge) when the truck is driving over it.

3. Custom Overlays

SnowRunner Editor allows to integrate custom overlays and use them for creating new maps together with the regular overlays.

Below you can find an overview of the process of creating a custom overlay. The sample files used for this overview can be found in the 9.2. Sample Overlays section.

There are two possible types of overlays: **texture** overlays and **geometric** overlays.

3.1. Texture Overlays

When creating a map, texture overlays are mostly used for various kinds of roads, i.e., objects which are flat and pressed to the terrain.

The subsections below give a brief overview of the process of creating a custom texture overlay.

3.1.1. Files of a Texture Overlay

Usage of a custom texture overlay in the Editor requires the following files in the following folders:

Folder	***Files***	***Comments***
Media\textures\overlays	**.tga** files for `Texture` and `TextureRelief` parameters	This folder should contain textures that are used in the ***<OverlayBrand>*** tag in the **.xml** file of the Class.
Media\prebuild\common	**.tga** files for `HeightmapOffset` and `SnowmapMask` parameters	This folder should contain textures that are used in the ***< Prebuild>*** tag in the **.xml** file of the Class.
Media\classes\overlays	**.xml** file of the class	The **.xml** file of the class of the overlay defines its properties: width, flattening, etc.

The **textures\overlays** and **prebuild\common** folders must be created in the **Media** folder, which is located in the **Documents\My Games\SnowRunner** folder.

3.1.2. Create necessary textures for an overlay

You can create and use any textures for your overlay. Nevertheless, they must follow certain rules below.

Names of textures: Names can contain Latin characters, digits, and underscores ("_"). They must contain no special characters (except "_") and no spaces.

Suffixes in names: Textures should have corresponding postfixes in their names, written after the original name of the texture. The typical texture set for an overlay should include a diffuse texture with an alpha channel, a diffuse texture of the relief with an alpha channel and optional textures for terrain height modifying with the following postfixes:

__postfix	***Parameter in XML of the mesh (in the <OverlayBrand> and <Prebuild> tags)***
__d_a	`Texture, TextureRelief`
__s	`HeightmapOffset`
__s_d	`SnowmapMask`

Channels of your textures should correspond to the following:

Channels	***Parameter in XML of the mesh (in the <OverlayBrand> and <Prebuild> tags)***	
	`Texture`	`TextureRelief`
RGB	Albedo Map	Normal Map
Alpha	Roughness Map	Height Map

Format: The files of textures must be in the **.tga** format.

Texture sizes: The dimensions of a texture should be $\mathbf{2^N\ x2^{N+1}}$. I.e., each side of the texture should be the power of two, and its width must be **twice the size** of its height. Recommended texture size for main overlay texture is **2048x1024**. Higher texture sizes for models might impact the performance or texture streaming in general.Textures for terrain height changing do not need to have high resolution, the recommended size for them is **128x64**.

Location: You should locate textures for your overlay according to the 3.1.1. Files of the Texture Overlay section of this guide.

3.1.3. Create the XML file of the Class for an overlay

Now, create the XML file of the class of the overlay in the **Media\classes\overlays** folder.

For example:

```
<OverlayBrand
     GenerateDust="true"
     Texture="overlays/us_asphalt_road_double_2__d_a.tga"
     TextureRelief="overlays/us_asphalt_road_double_2_relief__d_a.tga"
>
     <Prebuild
          DefaultWidth="8"
          FlattenPart="1"
          HeightmapOffset="hmoffset_dirt_road__s.tga"
          SnowmapMask="overlay_dirt_path_sn_01__s_d.tga"
          LandmarkColor="(100; 90; 80)"
          LandmarkPart="0.8"
          Layer="2"
</OverlayBrand>
```

To acquire more information about tags and attributes used for the creating of an overlay, please refer to the 8. Tags and Attributes of Overlays chapter below.

3.2. Geometric Overlays

Geometric overlays represent such three-dimensional objects as pipes, wires, borders etc. The process of the creation of a custom geometric overlay is quite similar to the one of a custom model.

3.2.1. Files of a Geometric Overlay

Usage of a custom geometric overlay in the Editor requires the following files in the following folders:

Folder	*Files*	*Comments*
Media\meshes\overlays	**.fbx** file of the mesh	The **.fbx** file of the overlay contains its mesh and its collision mesh.
	.xml file of the mesh	The **.xml** file of the mesh, stored in the same folder, contains data on the material of the overlay and its textures.
Media\textures\overlays	**.tga** files of textures	This folder should contain all textures that

		were assigned to the mesh of the overlay in the **.xml** file of the mesh.
Media\classes\overlays	**.xml** file of the class	The **.xml** file of the class of the overlay defines its properties: friction, overlay type, etc.

The **textures\overlays** and **prebuild\common** folders must be created in the **Media** folder, which is located in the **Documents\My Games\SnowRunner** folder.

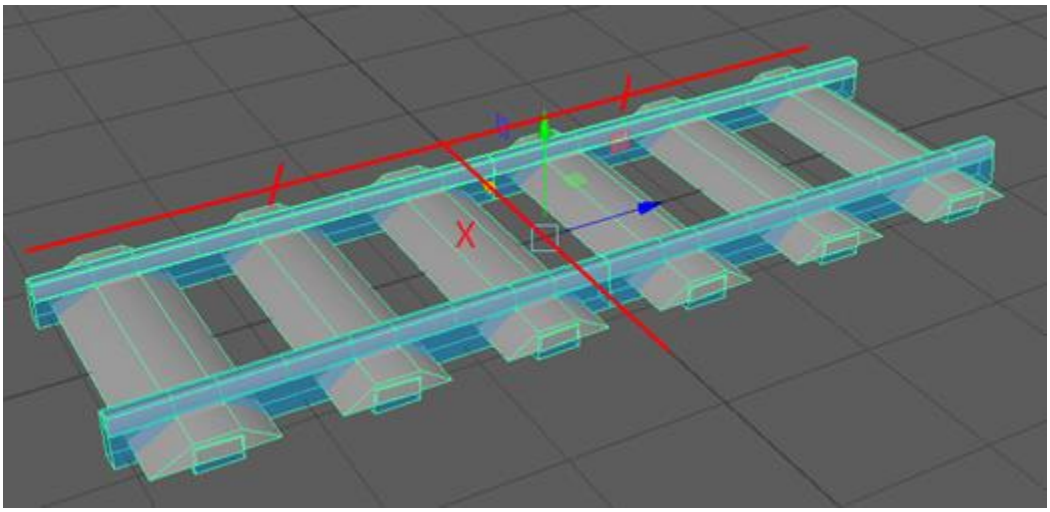
3.2.2. Create the model of an overlay in the 3D Package

The process of creating a model for an overlay is almost the same as described in the subsection 2.2.1. Create the model in the 3D Package. However, there are some additional rules, that must be adhered :

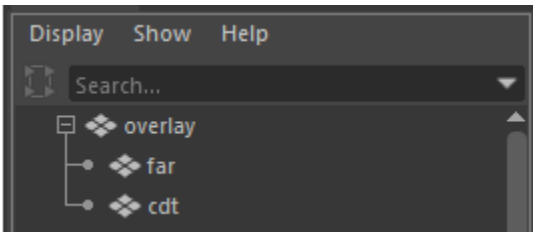
- All vertices and edges of the model must be welded together. Otherwise, when using a custom overlay in the Editor, gaps may appear between the overlay segments.

- The model you create must be symmetrical relative to the X coordinate axis (for Maya, 3DS Max and Blender 3D Packages).

  Sample model displayed in the 3D Package is shown below (red lines show the symmetry):



- The visible mesh must be called “**overlay**”, and the collision mesh – “**cdt**”. Optionally you can create additional mesh of the model for another LOD as the “**far**” object. The typical hierarchy of the model is shown below:



- The Editor supports only overlays containing a single material. For this reason, the only one material must be used when creating a model.

- If the type of the geometric overlay in properties of its class is specified as "**Geometric**" (for more information about types of geometric overlays, please refer to 8.1.1. <Prebuild> section), the overlay segments may be displayed *upside down* in the Editor.

  This type of overlays is mostly used for creating pipes, which are symmetrical, so that there is no difference between the display in the Editor and in the 3D Package. But if you want to create an asymmetrical model for the overlay of this type, note that the model should be turned upside down in a 3D Package to be displayed correctly in the Editor.

After creating the model, you need to export it to the FBX format in a similar way to exporting your mod of the truck to FBX. More detailed information can be found in the "**Exporting to Fbx: 3ds Max, Maya, and Blender**" guide, available at https://snowrunner.mod.io/guides/exporting-to-fbx-3ds-max-maya-and-blender.

After the export, you need to put the resulting FBX file into the **Media\meshes\overlays** folder.

The rules for naming FBX files for overlays are the same as described in the 2.2.2. Export a model to the FBX format chapter above, e.g., you can name it "**highway_border.fbx**".

3.2.3. Create textures for an overlay

The recommendations for creating textures for a geometric overlay are the same as for creating textures for a model (see 2.2.3 Create necessary textures for a model subsection), except for

Location: You should put textures to the **Media\textures\overlays folder**.

3.2.4. Create the XML file of the Mesh for an overlay

Create the XML file of the mesh for your FBX file in the **Media\meshes\overlays** folder. Names of the created XML file and the FBX file should be the same, except for the format, e.g., “**highway_border.xml**” for “**highway_border.fbx**”.

Here, you need to specify the links to all textures used by the model of an overlay and some other properties (see the example below). For details on tags and attributes used here, please refer to 8.2. <CombineXMesh> section.

Sample XML file of the mesh of the overlay:

```
<CombineXMesh
     Type="Overlay"
>
     <Material
          AlbedoMap="overlays/highway_border__d.tga"
          Name="highway_border_mat"
          NormalMap="overlays/highway_border__n_d.tga"
          ShadingMap="overlays/highway_border__sh_d.tga"
     />
</CombineXMesh>
```

3.2.5 Create the XML file of the Class for an overlay

Now, you need to create the XML file of the class for the overlay in the **Media\classes\overlays** folder.

You should name the created XML file as the FBX file and exactly as XML-mesh file, e.g., “**highway_border.xml**”.

In this file, you should specify the properties of the overlay. For example:

```
<OverlayBrand IsNotStaticSoil="true" BodyFriction="0.5" >
     <Prebuild OverlayType="Geometric" />
</OverlayBrand>
```

To learn more about tags and attributes used here, please see 8.1. <OverlayBrand> section below.

4. Custom PbrMaterials

Along with custom models and overlays you can create custom PbrMaterials, integrate them with the Editor and create new maps using them.

Below an overview of the process of creating a custom overlay is described. The sample files used for this overview can be found in the 9.3. Sample PbrMaterials section.

4.1. Files of a custom PbrMaterial

Usage of a custom PbrMaterial in the Editor requires the following files in the following folders:

Folder	*Files*	*Comments*
Media\textures\tiles	**.tga** files for `Texture` and `Normal` parameters	This folder should contain textures that are used in the ***<MaterialType>*** tag in the .xml file of the PbrMaterial.
Media\classes\terrain_layers	**.xml** file of the class	The **.xml** file of the class of the PbrMaterial defines its properties: types of grass, interaction sounds, viscosity, etc.

The **textures\tiles** and **classes\terrain_layers** folders must be created in the **Media** folder, which is located in the **Documents\My Games\SnowRunner** folder.

NOTE: As the interaction sounds, you can use only predefined sound files that are provided with the game. For details, see the `Sound`, `EndEffectSound`, and `SkidSound` attributes in 8.1. <MaterialType>.

4.2. Create necessary textures for a PbrMaterial

Textures you create for your PbrMaterial must follow rules below:

Texture set: The mandatory texture set for a PbrMaterial must include a diffuse texture with an alpha channel (`Texture`) and a diffuse texture of the relief with an alpha channel (`Normal`). However, additional textures may be included when creating a snowy layer (see the table below).

Parameter in XML of the mesh (in the <MaterialType> tag)	*Description of the texture*
`Texture`	The basic texture of your PbrMaterial.
`Normal`	The relief texture of your PbrMaterial.
`ReliefNormals1`	The additional texture for snowy layers. For example, in the “snow_layer” PbrMaterial this parameter provides an additional normal map describing the specifics of the forest snow: little bumps of snow covering small bushes and grass.
`ReliefNormals2`	The additional texture for snowy layers. In the “snow_layer” PbrMaterial this parameter provides an additional normal map describing the specifics of the snow along the roads: small packed clumps of snow.
`SnowDetailNormal`	The additional texture for snowy layers. In the “snow_layer” PbrMaterial this parameter provides a normal map that describes little details of snow surface: small granular particles of snow.

Texture channels: Channels of your textures should correspond to the following:

Channels	*Parameter in XML of the mesh (in the <MaterialType> tag)*	
	`Texture`	`Normal, ReliefNormals1, ReliefNormals2, SnowDetailNormal`
RGB	Albedo Map	Normal Map
Alpha	Roughness Map	Height Map

Names of textures: Names can contain Latin characters, digits, and underscores ("_"). They must contain no special characters (except "_") and no spaces.

Suffixes in names: All textures here should have the “__**d**_**a**” postfix in their names.

Format: The files of textures must be in the **.tga** format.

Texture sizes: Texture dimensions should be $\mathbf{2^N x 2^N}$. I.e., each side of the texture should be the power of two, and its width has to be equal to its height. Typical texture

size for models should not exceed **2048x2048**. Higher texture sizes for models might impact the performance or texture streaming in general.

Location: You should locate textures for your PbrMaterial according to the 4.1. Files of the custom PbrMaterial section of this guide.

4.3. Create the XML file of the Class for a PbrMaterial

Now, create the XML file of the class of the PbrMaterial in the **Media\classes\terrain_layers** folder.

For example:

```
<MaterialType
	AllowHiddenExtrudes="0.25"
	Texture="tiles/grass_rus_02__d_a.tga"
	Normal="tiles/grass_rus_02_relief__d_a.tga"
	GrassBrush="GrassShortRusA,GrassWeedRusA,GrassWeedRusD"
	IsThickGrass="false"
	Sound="surfaces/surf_grass_mix_loop"
	EndEffectSound="surfaces/hit/hit_grass_rnd"
	SkidSound="surfaces/skidding/skidding_grass_loop"
	MinViscosity="2.0"
/>
```

PbrMaterials which can be painted with the **“Snow”** brush require some additional parameters. You can learn more about snowy PbrMaterials from the **5.2.7. Snow** and the **5.3.1. Material properties** sections of the “**SnowRunner™ Editor Guide”** available as the **“SnowRunner_Editor_Guide.pdf”** of the same documentation package.

Sample Class for the predefined PbrMaterial “snow_layer”:

```
<MaterialType
	AllowHiddenExtrudes="1"
	MinViscosity="2"
	NoGrass="false"
	Normal="tiles/snow_01_relief__d_a.tga"
	ReliefNormals1="tiles/snow_forest_relief__d_a.tga"
	ReliefNormals1BlendContrast="0.7"
	ReliefNormals2="tiles/snow02_relief__d_a.tga"
	ReliefNormals2BlendContrast="0.3"
	ReliefNormals2BlendFactorAtNight="0.4"
	ShadingType="snow"
	SnowAlbedoColor="(230; 230; 230)"
	SnowDetailFactor="0.25"
	SnowDetailNormal="tiles/snow_detail__n_d.tga"
```

```
        SnowDetailTiling="3.0"
        SnowTintBrightness="0.55"
        Texture="tiles/snow_01__d_a.tga"
        UpVectorSnowAlbedoColor="(210; 210; 210)"
        WetnessAlbedoMultiplier="0.1"
        WetnessRoughnessMultiplier="0.1"
        Sound="surfaces/surf_snow_rnd"
        SkidSound="surfaces/skidding/skidding_snow_loop"
        IsSnowParticles="true"
/>
```

To learn more about tags and attributes used for the creating of a PbrMaterial, please refer to the 8.1. <MaterialType> section below.

5. <_templates>

This tag allows to inherit some code from templates.
For info on this feature, see the "***5.6. Templates (_templates)***" section of the **Integration of Trucks and Addons** guide (available at https://snowrunner.mod.io/guides/integration-of-trucks-and-addons-part-2).

In the XML descriptions of models and brushes, we cannot create custom templates and can only use the existing ones, stored in the **initial.pak** archive, in the **[media]_templates** folder.

6. Tags and Attributes of Models

The sections below provide some info on main tags and attributes used in XML descriptions of models.

The hierarchy of subsections in these sections corresponds to the hierarchy of tags.

6.2. <CombineXMesh>

The root tag of the XML file of the Mesh.

In general, this tag is a root tag in all XML files of meshes (i.e., of model, plant, truck, etc.). However, this section describes only tags related to models.

Attributes:

- `Type="Model"`
  The "**Model**" value of this attribute specifies that this XML file of the mesh corresponds to a model.

6.2.1. <MeshLod>

Allows you to set up distances for LODs of your model.
This tag can be used multiple times if you want to have different LOD settings corresponding to different meshes (see `MeshName` below).

Attributes:

- `Distances="15, 350"`
  Distances from camera to model, in meters, with comma as a delimiter. These distances are used for switching from LOD to LOD for a model.
  Number of these distances = Number of LODs of the geometry – 1.
  For plants, the system always generates the last LOD (billboard) automatically, and it should be taken into account during the setup of LOD settings.

- `HidingDistance="500"`
  Distance from the camera, at which the model will be hidden (will not be rendered).

  NOTE: The final lodding distances and the hiding distance of a particular *instance* of the model will also depend on the **Scale** of this instance. The higher the scale of the object, the greater the resulting distance at which it will disappear.

- `MeshName="antenna"`
  Optional. If it is used, LOD settings are applied to the specified mesh. If it is not used, LOD settings are applied to all meshes of a model.

6.2.2. <MeshShadow>

Allows you to control the distance for displaying the dynamic shadow of the model.

Attributes:

- `HidingDistance="50"`
  Distance from the camera, in meters. When the distance between the camera and the model is less than the specified value, the dynamic shadow from this model will be rendered.

6.2.3. <Material>

Similar to the `<Material>` tag used for trucks.

See the "***6.2. <Material>***" section of the **Integration of Trucks and Addons** guide available as the **"Integration_of_Trucks_and_Addons.pdf"** of the same documentation package.
The same properties of the material can be used here.

For each material, in the **Name** attribute, you need to specify the exact name of the material from the FBX file.
When defining textures of this material (`AlbedoMap`, `NormalMap`, etc.) you need to specify paths to the textures that you put into the **Media\textures\models** folder. The paths here should be specified relative to the **Media\textures** folder.

However, for models, the `<Material>` tag can contain the `<ModelMaterial>` tag, see below.

6.2.3.1. <ModelMaterial>

Allows to set material properties specific for the model.

Attributes:

- `IsWheelTracks="false"`
  If enabled, trucks will be able to leave tracks on the material of the model.

- `NoOcclusion="false"`
  If enabled, disables occlusion on the material of the model that comes from other models.

6.3. <ModelBrand>

The root tag of the XML file of the class of the model.

Attributes:

- `ClipCamera="true"`
  Defines whether or not the camera can move inside the model. The camera can work in two modes:
  - `ClipCamera="true"` - The camera is raised above the model. This is the default mode.
  - `ClipCamera="false"` - The camera ignores the model and is able to move inside the geometry of the model. This mode is used for road signs, fences, poles, and so on.

- `DynamicModel="false"`
  Marks this model as “dynamic”. See 2.3.1.2. Rule #2: DynamicModel parameter.

- `Instanced="false"`
  Allows you to enable instanced rendering (instancing) for the model. See 2.3.1.1. Rule #1: Attributes for Instanced Rendering for details.

- `InstancedAsModel="false"`
  Allows you to enable instancing for the *static* model. See 2.3.1.1. for details.

- `AcceptZoneBorderDecals="false"`
  If this option is enabled, the game will be able to draw the borders of various game zones (zone for loading cargo, Garage entrance, zone of delivery, and so on) on the surface of this model. This option is important for large models that may be located on the ground near game zones.

- `CastDayLightmapShadow="true"`
  If this option is enabled, the static lightmap shadows are enabled for this model during the day. Static lightmap shadows are rendered at the distance greater than 50 meters from the camera. At the distance that is less than 50 meters - dynamic shadows are used. For more details, see the description of the **Sun Static Direction** field in “*5.1.1. Terrain Properties*” in **“SnowRunner™ Editor Guide”** available as the **“SnowRunner_Editor_Guide.pdf”** of the same documentation package. By default, this option is “true”.

- `CastNightLightmapShadow="true"`
  If this option is enabled, the static lightmap shadows are enabled for this model during the night. Static lightmap shadows are rendered at the distance greater than 50 meters from the camera. At the distance that is less than 50 meters - dynamic shadows are used. For more details, see the description of the **Sun Static Direction** field in “*5.1.1. Terrain Properties*” in **“SnowRunner™ Editor Guide”** available as the **“SnowRunner_Editor_Guide.pdf”** of the same

documentation package .
By default, this option is “true”.

- `ClipCameraBridge="false"`
  Allows you to set up the specific behavior of the camera for bridges.
  If enabled, the game engine will not raise the camera upwards (above the bridge) when the truck is driving over it.

- `ApplyNormalInReference="false"`
  Defines model behavior as part of reference. Particularly, whether or not the model should be inclined if its parent reference has been added to the uneven terrain of the target map.

- `BreakType="SmallTree"`
  Here you can specify the name of the BreakType that will be played at the moment when this model becomes broken (when its constraints are disabled). The BreakType is a combination of an effect and a sound. All BreakTypes used by the game are described in the **BreakTypes.xml**, which is stored in the **initial.pak** archive at the following path:
  [media]\classes\media_data_types\BreakTypes.xml
  However, the setup of a BreakType is a topic for the separate document.

- `InBackground="false"`
  Whether or not the model is rendered in the background (e.g. as a farplane). Lightmaps are not calculated for such objects.

6.3.1. <PhysicsModel>

Section that defines the physical properties of the model: physical bodies it consists of and their interactions with themselves and the environment.

NOTE: For trucks, we were required to specify the path to the XML file of the mesh in the mandatory **Mesh** attribute of the **<PhysicsModel>** tag in the XML file of the class. However, for models, this attribute is not used. See 2.1.1. Important Note on File Naming.

Below you can find some notes on the setup of physical properties for static and dynamic models. For differences between these types, see 2.3.0. Types of Models.

WARNING: Every model must have at least one cdt mesh.

For static models, the general setup of physical properties is the following:

1. ***A static model has one physical body and one cdt mesh.***
   A static model can have only one physical body (**<Body>** tag). If multiple **<Body>** tags are specified, only the first one is taken into account, all other **<Body>** tags are ignored.
   This single **<Body>** tag should correspond to a single cdt mesh. If multiple separated collision objects are needed, they should be united in a single cdt mesh consisting of a necessary number of non-intersecting parts.
   As with dynamic models, the relation of the body to the cdt mesh is set with the help of a frame of a model in the FBX file, which can be specified in the **ModelFrame** parameter of the **<Body>** tag. The system will search for a parent frame of the cdt mesh (with name starting with **cdt**_) among the direct children of the specified frame.
   However, most static models typically have the parent frame of the cdt mesh as a direct child of the root node and the game engine can find it automatically. Thus, in this case, the **ModelFrame** parameter of the **<Body>** tag can be omitted (as in the sample below).

2. ***Most models of this type act as simple collision objects.***
   Physical bodies of static models have no mass. Typically, these models act simply like static collision objects. Thus, `Collisions="Dynamic"` needs to be specified for them (see description of this attribute in 6.3.1.1. <Body>).

   So, for most of static models, descriptions of their physics model is fairly simple:

```
<PhysicsModel>
      <Body Collisions="Dynamic" />
</PhysicsModel>
```

   However, if necessary, collisions can be disabled also.

For dynamic models, the general setup of physical properties is the following:

1. ***A model has physical bodies connected to model frames and cdt meshes***.
   Each model can contain a certain number of physical bodies (**<Body>** tags) that correspond to some of the frames of the model in the FBX file. I.e., each body corresponds to a frame, which can be the bone, the mesh, or the transform object (in the game engine terms). The relation of the physical body to a frame is set by the **ModelFrame** parameter of the **<Body>** tag, where the name of the frame is specified. Among direct children of these frames in the FBX hierarchy, the system searches for collision meshes (their names start with **cdt**_). I.e., for each frame mentioned in a **<Body>** tag, the system tries to find a corresponding cdt mesh.
   Physical bodies of a model can have a hierarchy, i.e. **<Body>** tags can be nested within each other. There can be only one **<Body>** tag on the highest level of hierarchy. However, each **<Body>** tag can have as many child **<Body>** tags as necessary. The hierarchy of **<Body>** tags may correspond to the hierarchy of frames in the FBX file or it can be defined independently. However, each body should correspond to a frame with a cdt mesh.

   NOTE: A dynamic model can have some static parts. I.e., a dynamic model can contain some static physical bodies with zero mass, which can be connected to other dynamic physical bodies of the model. In a static model, by default, all meshes are associated with a single static physical body with a zero mass. In dynamic models, there can be multiple physical bodies and they can have non-zero mass.

2. ***Constraints bind the physical bodies of the model.***
   Physical bodies of a model are connected by constraints that define *how* they are connected (e.g. whether they are on a fixed, motionless connection, or they can move and rotate, and so on). This is done in a **<Constraint>** tag. When a **<Body>** tag contains a child **<Body>** tag, these bodies must be connected by a constraint of some type.
   Since **<Body>** tags correspond to cdt meshes, constraints define how these cdt meshes are connected to each other. E.g., if one of the connected meshes experiences a collision, the physics system will calculate its movement and rotation in this collision based on the constraint.
   Moreover, a constraint can connect not only the internal bodies of the model but the root body of the model *to the world*. I.e. using the constraint you can bind the model to the point where this model is located in the Editor. Typically, for this type of constraint, you will specify the `BreakOffThreshold` too, see below.

3. ***Constraints can break (`BreakOffThreshold`).***
   All constraints specified for the model can be broken. This is done by specifying the appropriate value of the **`BreakOffThreshold`** attribute of the **<Constraint>** tag. This parameter defines the threshold for the force applied to the model before breaking this constraint. If the applied force is greater than this threshold, this constraint of the model will become broken and will stop binding its bodies.

This `BreakOffThreshold` is frequently used for constraints that bind the model to the world (e.g. it can be used for a road sign that will be placed on some locations on the map). We recommend this approach to avoid incorrect behavior of the model when a truck drives near it - see 2.3.2.1. Main Rules and 6.3.1.1.1. <Constraint> for details.

4. ***Constraints can have an internal "movement muscle" (Motor).***
   Constraints specified for physical bodies of a model (or body VS world) can have a "movement muscle", which is described in a **<...Motor>** tag within this constraint. This motor tries to stop the movement of the physical body, which is allowed by constraint (e.g. as a spring placed in the joint between two bodies). If a motor is not described, then the connected body will hang loose under the influence of gravity and inertia according to the limitations of the constraint.
   Please note that by **<...Motor>** tag we actually mean one of the following tags: 6.3.1.1.1.1. <Motor>, 6.3.1.1.1.2. <PlaneConeMotor>, and 6.3.1.1.1.3. <AllMotor>.

6.3.1.1. <Body>

A physical body.
A physical body is the combination of the attachment point and the collision mesh that participates in Havok physics. The physical body can collide with other physical bodies. The physical body can be attached to its parent by various means: move linearly relative to its parent within some limits, or swing along one or several axes, or be fixed tightly, etc. The physical body has some physical characteristics: mass, friction, and so on.

The **<Body>** tags can contain other **<Body>** and **<Constraint>** tags, forming the hierarchy of physical bodies of the model and the connections between them. The hierarchy of the bodies in the description of the physics model can correspond to the hierarchy of bones in the FBX file or it may be different.
In any case, this hierarchy must have ***only one root physical body***. However, each **<Body>** tag may contain multiple other **<Body>** tags.

Each physical body must have a corresponding cdt frame. The relation between them is defined with the help of the appropriate frame of the model:

- The physical body corresponds to a frame from the FBX file (this relation can be explicitly set by the **ModelFrame** parameter of the **<Body>** tag or can be set implicitly if this parameter is omitted).
- Among direct children of this frame in the FBX hierarchy, the system searches for a collision mesh (its name should start with **cdt**_). I.e., for each frame mentioned in a **<Body>** tag, the system tries to find a corresponding cdt mesh.

Physical bodies of a model are connected by constraints that define *how* they are connected (e.g. whether they are on a fixed, motionless connection, or they can move and rotate, and so on). This is done in a **<Constraint>** tag (see 6.3.1.1.1. <Constraint>). When a **<Body>** tag contains a child **<Body>** tag, these bodies must be connected by a

constraint of some type.

For static models, there should be only one physical body. This body will have no mass and will act like a static collision object from the physical point of view. This body must correspond to a single cdt mesh. Moreover, most static models typically have the parent frame of the cdt mesh as a direct child of the root node and the game engine can find it automatically. Because of that, the **ModelFrame** parameter of the **<Body>** tag can be omitted for them. So, for most of the static models, descriptions of their physical body is fairly simple:

```
<PhysicsModel>
      <Body Collisions="Dynamic" />
</PhysicsModel>
```

For details, please refer to the 6.3.1. <PhysicsModel>.

For dynamic models, the configuration is more complicated. There can be multiple physical bodies with the non-zero mass and multiple corresponding cdt meshes. These bodies must be connected to each other or “to the world” using constraints, which can be breakable and can have a motor affecting the constraint movement. All these factors will be taken into account when the model will move (according to Havok physics). For more details, please refer to the 6.3.1. <PhysicsModel>.

Attributes:

- `Collisions="Dynamic"`
  Defines the collision model (layer) that will be used for the model.
  For most of the models, we need to set `Collisions="Dynamic"`. This means that the solid body of this model has collisions with other models with `Collisions="Dynamic"`. For example, this is valid for buildings that have the barrels and other objects near them and for these barrels and objects too.
  However, for models with breakable constraints, you need to take into account that such objects will have no collision with the ground surface until any of their constraints becomes broken (see 2.3.2.1. Main Rules for Physical Models above).
  Another option here is `"None"`, which can be specified if you want no collisions at all.

- `Friction="0.9"`
  Allows you to specify the friction coefficient that will be used if the (body of the) model contacts another physical body (e.g. a truck, another model, and so on).

- `Mass="100"`
  Mass of the physical body. Static models cannot contain solid bodies with non-zero mass. Dynamic models contain at least one (or more) physical body with non-zero mass.

- `NoSoftContacts="false"`
  For most of the physical (dynamic) models, we need to set this option to `"true"`. This is necessary for hard contacts of models with trucks, to avoid the situation when wheels of the truck move inside the model.

- `ModelFrame="BoneRoot"`
  Allows you to specify the frame (a bone, a visual mesh, or a transform object) in the FBX file that is a parent of the cdt mesh for this body. For details, see 6.3.1. <PhysicsModel> or the description of the **<Body>** tag above.
  For most static models, this attribute is omitted, since the parent bone of the cdt mesh there is typically created as a direct child of the root node and the game engine can find it automatically.

- `CenterOfMassOffset="(0; -1.6; 0)"`
  The shift of the center of mass of the model, relative to the default center of mass. See 2.3.2.1. Main Rules.

- `IsCapsuleCDT="false"`
  Enabling this option forces the system to use capsules as physical shapes instead of the initial geometry of cdt mesh. The capsule is created automatically and roughly covers the volume of the initial cdt. The usage of capsules results in smoother collisions and is cheaper from the point of view of the performance.

- `IsStaticSoil="false"`
  If enabled, the model surface is considered the soil and, in the game, some pieces of this soil will be thrown from under the wheels of a truck moving on this surface.

- `IsGameAsphalt="false"`
  If this option is enabled, the surface of this model will be marked as “asphalt” for various parts of the game logic.

- `DamageMult="1.0"`
  This coefficient allows you to adjust the amount of damage received by a truck when it hits this physical body. Frequently, this coefficient is used to completely disable the damage from this physical body if this damage is not necessary. In this case, this coefficient is set to zero (`DamageMult="0.0"`).

- `AngularDamping="10"`
  The coefficient of reduction of angular momentum by time. It allows you to slow down rotating bodies (as it happens in the real world due to the presence of the atmosphere and other factors).

6.3.1.1.1. <Constraint>

A way to bind a physical body to its parent body. I.e., a constraint connects two physical bodies to each other. Or, it can bind a physical body “to the world” (i.e. to the point where the model is located on the map).

A constraint can have a child **<...Motor>** tag that works as a “movement muscle” for this connection. Please note that by **<...Motor>** tag we actually mean one of the following tags: 6.3.1.1.1.1. <Motor>, 6.3.1.1.1.2. <PlaneConeMotor>, and 6.3.1.1.1.3. <AllMotor>.

NOTE: Constraints are used for dynamic models only.

Usage scenarios for the **<Constraint>** tag are the following:

1. ***Adding a constraint between a parent physical body and a child physical body***.
When a **<Body>** tag contains a child **<Body>** tag, these bodies *must* be bound by a constraint, which will specify how these bodies are connected. In this case, we need to specify the constraint for these bodies as a **<Constraint>** tag that is a child tag of the *second* body. Please note that we need to put the **<Constraint>** tag inside the *child* **<Body>** tag, *not* into the parent **<Body>** tag.
I.e. the tag pattern here will be the following:

```
<PhysicsModel>
      <Body ... ModelFrame="Bone_1">
            <Body ... ModelFrame="Bone_2">
                  <Constraint ... />
            </Body>
      </Body>
<PhysicsModel>
```

Where the constraint above binds the parent body (attached to `Bone_1`) to the child body (attached to `Bone_2`).

2. ***Adding an extra constraint and binding two physical bodies (two frames) that have no parent-child relation between them***.
In this case, the **<Constraint>** tag is written *not* as a child tag of the **<Body>** tag, but as a direct child of the **<PhysicsModel>** tag. To specify target frames we want to bind, we use **`ModelFrameParent`** and **`ModelFrameChild`** parameters of the **<Constraint>** tag.
I.e. the tag pattern here will be the following:

```
<PhysicsModel>
      <Constraint
            ...
            ModelFrameChild="Bone_1"
            ModelFrameParent="Bone_15"
      />
      ... (some <Body> tags hierarchy)
</PhysicsModel>
```

3. *Binding a model (its root physical body) to the world*.
In this case, we need to put the **<Constraint>** tag inside the root **<Body>** tag. I.e., the child body here will be this root **<Body>** tag and the parent will be *the world* (the point where the model is located on the map).
I.e. the tag pattern here will be the following:

```
<PhysicsModel>
	<Body	... ModelFrame="Bone_1">
		<Constraint ... />
		... (if necessary, some <Body> tags hierarchy)
	</Body>
</PhysicsModel>
```

Where the constraint above binds the root parent body (attached to `Bone_1`) to the world.
Typically, the `BreakOffThreshold` attribute is also specified for constraints that bind the model to the world. We recommend this approach to avoid incorrect behavior of the model when a truck drives near it - see 2.3.2.1. Main Rules for details.
For example, this way the **barrel_02** model is bound to the world (code below does not use templates for convenience, original code uses them):

```
<PhysicsModel>
	<Body
		Mass="200"
		NoSoftContacts="True"
		Collisions="Dynamic"
		ModelFrame="BoneRoot"
	>
		<Constraint
		BreakOffThreshold="100"
		Type="Fixed"
		/>
	</Body>
</PhysicsModel>
```

This code allows a barrel to avoid movement until it is slightly pushed by a truck.

Attributes:

- `BreakOffThreshold="500"`
  Threshold for the force applied to the model. If the applied force is greater than this threshold, this constraint of the model will become broken and will stop binding its bodies. See 2.3.2.1. Main Rules.

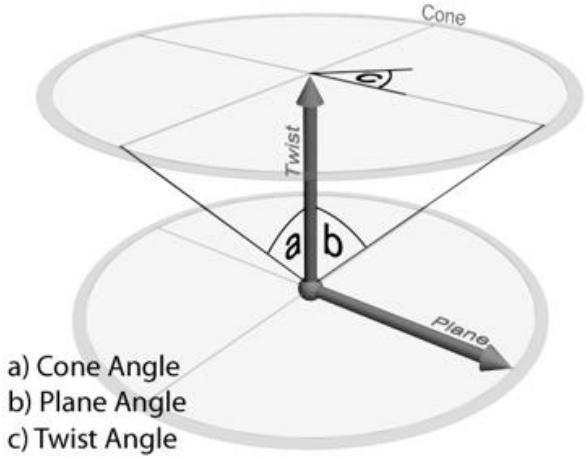
- `Type="Fixed"`
  The type of the constraint:
  - **Fixed** - fixed connection (constraint). Describes motionless, fixed connection.

    - **Hinge** - Rotation by one axis within specified limits. This type of constraint is suitable for describing a door hinge.
    - **UnlimitedHinge** - unlimited rotation along a given axis.
    - **Ragdoll** - Rotation by all axes within specified limits. The connection is suitable for describing the shoulder joint or bell tongue.
    - **Prismatic** - linear movement along one axis within specified limits. This type of constraint is suitable for the description of a pump, hydraulics, and so on.

- `PivotOffset="(1; 0.1; 0)"`
  Offset of the Pivot of the constraint. I.e., offset of the attachment point of the constraint relative to the pivot of the parent physical body. Rotations of constraints of the Hinge and Ragdoll types are performed relative to this point.

- `AxisLocal="(0; 0; 1)"`
  The direction vector of the rotation axis for constraints of **Hinge** and **UnlimitedHinge** types. Default value: (0; 1; 0).

- `PlaneAxisLocal="(1; 0; 0)"`
  The direction of the **Plane** axis (see picture below), which is perpendicular to the **Twist** axis, for the **Ragdoll** type of constraint. Default value: `(1; 0; 0)`.
  For unambiguous determination of this type of constraint, it is necessary to specify two perpendicular direction vectors. The third vector (**Cone**, see picture below) is unambiguously determined by the vector multiplication of **TwistAxisLocal** x **PlaneAxisLocal**.



- `TwistAxisLocal="(0; 1; 0)"`
  The direction of the **Twist** axis (see picture above) for the **Ragdoll** type of constraint. Default value: `(0; 1; 0)`. If we describe the joint of the shoulder, then this is the axis of rotation of the forearm.

- `TwistMin="-50"`
  Minimum angle of rotation along the **Twist** axis for the **Ragdoll** type of

constraint, in degrees. By default: 0, limits: [-180, 180].

- `TwistMax="70"`
  Maximum angle of rotation along the **Twist** axis for the **Ragdoll** type of constraint, in degrees. By default: 0, limits: [-180, 180].

- `MinLimit="-14"`
  The lower bound for movement used for the **Hinge** and **Prismatic** types of constraints.
  Default value:
  - Hinge: -360, limits [-360, 360].
  - Prismatic: 0

- `MaxLimit="10.1"`
  The upper bound for movement used for the **Hinge** and **Prismatic** types of constraints.
  Default value:
  - Hinge: 360, limits [-360, 360].
  - Prismatic: 0.

- `ModelFrameParent="joint_1"`
  The name of the parent frame of the constraint in the FBX file. This attribute is used when the **<Constraint>** tag is used outside the **<Body>** tag. I.e., when you need to specify an additional constraint between two frames.

- `ModelFrameChild="joint_15"`
  The name of the child frame of the constraint in the FBX file. This attribute is used when the **<Constraint>** tag is used outside the **<Body>** tag. I.e., when you need to specify an additional constraint between two frames.

- `Cone="15"`
  The maximum angle of the rotation of the bone inside the cone with the **Twist** axis (`TwistAxisLocal`) in the center. This parameter is used for the **Ragdoll** type of constraint and is specified in degrees. If you want to set up a rotation inside an asymmetrical cone, you can use the `ConeMin` and `ConeMax` parameters instead (see below).

- `ConeMin="-70"`
  Minimum angle of rotation along the **Cone** axis for the Ragdoll type of constraint, in degrees. By default: -180, limits: [-180, 180]. This attribute is used only if the `Cone` attribute is not set.

- `ConeMax="70"`
  Maximum angle of rotation along the **Cone** axis for the Ragdoll type of constraint, in degrees. By default: 180, limits: [-180, 180]. This attribute is used only if the `Cone` attribute is not set.

- `PlaneMin="-50"`
  Minimum angle of rotation along the **Plane** axis for the Ragdoll type of constraint, in degrees. By default: -180, limits: [-180, 180].

- `PlaneMax="70"`
  Maximum angle of rotation along the **Plane** axis for the Ragdoll type of constraint, in degrees. By default: 180, limits: [-180, 180].

6.3.1.1.1.1. <Motor>

Motor (“movement muscle”).

If the Body is a bone, the Constraint is the joint between the bones, then the Motor will be the muscle. If the Motor is not described, then the bone will hang loose under the influence of gravity and inertia according to the limitations of the Constraint.
Motor is not used with the constraints of the following types: Ragdoll, UnlimitedHinge, and Rigid.

Attributes:

- `Type="Spring"`
  Type of the motor. For models, we use only the “`Spring`” type, which corresponds to a spring.
- `Spring="0.5"`
  The stiffness of the spring. Default value: 0, limits: [0, 1000000000].
- `Damping="0.02"`
  The coefficient of damping.

6.3.1.1.1.2. <PlaneConeMotor>

Motor for describing rotation along the **Plane** and **Cone** axes.

This tag is used for the **Ragdoll** type of the constraint. All attributes are the same as in **<Motor>** (see above).

6.3.1.1.1.3. <AllMotor>

Motor for describing rotation along all axes.

This tag is used for the **Ragdoll** type of the constraint. All attributes are the same as in **<Motor>** (see above).

6.3.1.2. <Constraint>

Can be specified without the parent **<Body>** tag when it is necessary to specify an additional constraint between the physical bodies (frames). In this case, bound bodies are specified using **`ModelFrameParent`** and **`ModelFrameChild`** attributes.
See 6.3.1.1.1. <Constraint> for details.

6.3.1.3. <Flare>

Flare. A point source of light that creates a glow around the particular point. Visually similar to the bright light of a bulb or the distant light from the headlights in the face.

Attributes:

- `Pos="(3.759; 1.169; 0.944)"`
  Light source position.

- `Dir="(1; -0.3; 0)"`
  Direction vector.

- `DirAngle="90"`
  The angle of visibility of the flare, in degrees.

- `AttenStart="10"`
  The start of attenuation of the brightness of the flare, in meters. By default: 60.

- `AttenEnd="50"`
  Maximum length on which the flare is visible. By default: 120.

- `Color="g(255; 186; 112) x 2"`
  Light color and brightness multiplier. Default value: "(0; 0; 0)".

- `ColorMultAtDay="0.5"`
  A coefficient of the light brightness during the daytime.
  By default: 1, Values: [0; 1000].

- `Size="0.2"`
  The size of the flare. By default: 1, Values: [0.0001; 1000000].

- `SizeAtDay="0.2"`
  The size of the flare in the daytime.

- `SizeMultAtDay="1.0"`
  The intensity of the flare in the daytime. Frequently, the `0` value is used to disable the flare in the daytime.

- `OcclusionOffset="-0.4"`
  The offset of the flare towards the camera to avoid overlapping with the geometry of the model (the “towards” direction is set by negative values).
  For example, to avoid the flare of the lamp being covered by the lampshade, if the flare is put inside the lamp.

- `Texture="sfx/flare_simple__s_d.tga"`
  Path to the texture file of the shape of the flare, relative to the **Media\textures** folder. Default value: "sfx/flare_simple__s_d.tga".

- `AspectRatio="1.4"`
  The aspect ratio for horizontal and vertical sides of the sprite of the flare.

6.3.2. <Occlusion>

This tag allows you to apply an occlusion texture to the ground surface near your model in the Editor.

Attributes:

- `Size="(79.0; 62.0)"`
  Z and X coordinates that define the size of the ground surface near the model where the texture is applied.

- `Texture="occlusion_factory01.tga"`
  The name of the occlusion texture. TBD: Usage of custom occlusion textures is currently in the process of development. In-game occlusion textures are stored in the **editor.pak** archive, within the **[prebuild]\common** folder.

- `Intensity="0.5"`
  Intensity coefficient for the occlusion texture. Possible values are in the [0, 1] range.

6.3.3. <SnapPoint>

Snapping point that allows you to connect other objects to the model when working with models inside the Editor.
For example, you can connect wires to the lamp pole (see “*5.8.5. Wires: Adding and connecting them*” in “**SnowRunner™ Editor Guide”** available as the **“SnowRunner_Editor_Guide.pdf”** of the same documentation package).
This tag can be used multiple times.

Attributes:

- `Pos="(-0.729; 7.733; -0.169)"`
  Position of the snapping point.

6.3.4. <GameData>

Info on the interaction of the model with the environment (various gameplay data).

Attributes:

- `LoadType="CargoWoodenPlanks"`
  This parameter is used for models used as cargo objects. It defines the type of the cargo. This type is defined in a separate XML class with the same name (e.g. **CargoWoodenPlanks.xml**). This class contains links to UI descriptions and icons used for this type of cargo. The set of these CargoType classes used by the game can be found in the **initial.pak** archive, within the **[media]\classes\cargo_types** folder. The creation of new CargoTypes is currently not supported.

- `PackSlotsNumber="1"`
  This parameter is used for models used as cargo objects. It defines how many cargo slots this cargo will occupy.

- `LoadAddons="load_logs_short"`
  This attribute is not used anymore. It does not affect anything.

6.3.4.1. <WinchSocket>

Allows you to set a point on the model for attaching the winch.

Attributes:

- `Pos="(2.729; 7.733; 2.169)"`
  Position of the attachment point.

6.3.4.2. <CraneSocket>

Allows you to set a point on the model for grabbing it with the crane.

Attributes:

- `Pos="(2.729; 7.733; 2.169)"`
  Position of the attachment point.

6.3.5. <Landmark>

Allows you to set up how your model will be displayed on the navigation map.

Attributes:

- `Mesh="landmarks/factory01_lmk"`
  Path to the XML-mesh of the landmark model, relative to the **Media\meshes** folder and without filename extension.

  NOTE: To add a landmark model, which is displayed at the navigation map for your model, you need to create and add the lowres model in the FBX format (the **.fbx** file) and its XML-mesh with the same name (the **.xml** file) to **Media/meshes/landmarks/** folder. For more details, see 2.2.7. Optional: Landmark model.

6.3.6. <Subset>

This tag is related to the game logic of _**objective** models and is out of the scope of this guide. There will be a separate guide on this topic.

NOTE: For details on using *in-game* _**objective** models when creating maps, see see "***5.15.3. Model Building Settings***" in "**SnowRunner™ Editor Guide"** available as the **"SnowRunner_Editor_Guide.pdf"** of the same documentation package.

6.3.7. <AnimSubset>

This tag is related to the game logic of _**objective** models and is out of the scope of this guide. There will be a separate guide on this topic.

NOTE: For details on using *in-game* _**objective** models when creating maps, see see "***5.15.3. Model Building Settings***" in "**SnowRunner™ Editor Guide"** available as the **"SnowRunner_Editor_Guide.pdf"** of the same documentation package.

6.3.8. <TrackEvents>

This tag is related to the game logic of _**objective** models and is out of the scope of this guide. There will be a separate guide on this topic.

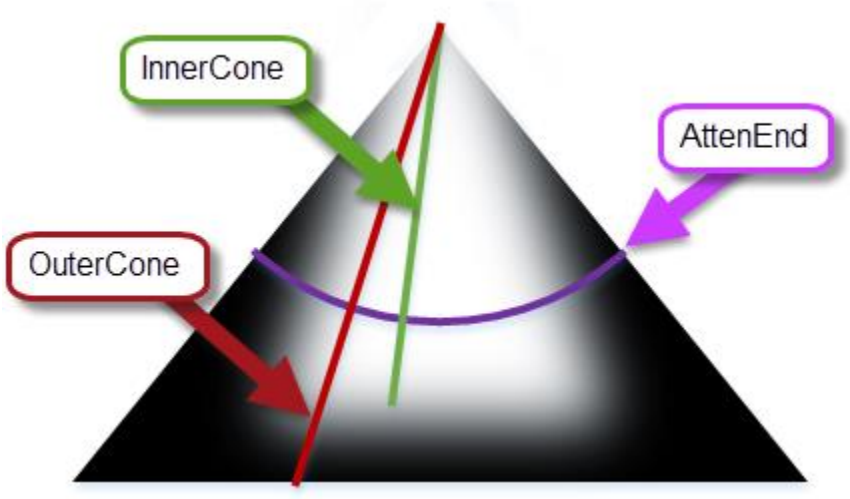
NOTE: For details on using *in-game* _**objective** models when creating maps, see see "***5.15.3. Model Building Settings***" in "**SnowRunner™ Editor Guide"** available as the **"SnowRunner_Editor_Guide.pdf"** of the same documentation package.

6.3.9. <StaticLights>

Section that contains one or more **<StaticLight>** tags, see below.

6.3.9.1. <StaticLight>

A source of the light that illuminates objects and surfaces in a cone of a light beam.



Attributes:

- `Pos="(-0.729; 7.733; -0.169)"`
  The position of the light source.
- `Dir="(1; -0.3; 0)"`
  The direction vector.
- `AttenEnd="50"`
  The maximum length at which the light illuminates objects. After this distance, the light diminishes quickly.
- `Color="g(255; 186; 112) x 2"`
  The light color and brightness multiplier.
- `CreateSuperLight="true"`
  If this option is enabled, this means that this light is able to work as a dynamic source of light, when a truck is located near it. I.e., at night this light will be able to cast shadows (if they are enabled), and so on.
  If this option is disabled, this will result in baking the light from this light source into the lightmap of the terrain and in absence of dynamical properties for this light (e.g. there will be no shadows from objects).
- `InnerCone="20"`
  The inner cone (where light does not scatter). By default: 360.
- `OuterCone="100"`
  The outer cone (where light scatters). By default: 0.
- `Shadows="false"`
  Enables shadows for the light.

7. Tags and Attributes of Overlays

The sections below provide some info on main tags and attributes used in XML descriptions of overlays.

The hierarchy of subsections in these sections corresponds to the hierarchy of tags.

7.1. <OverlayBrand>

The root tag of the XML file of the class of the overlay.

Attributes for texture overlays:

- `Texture="overlays/texture_name__d_a.tga"`
  Defines the basic texture of the texture overlay.
- `TextureRelief="overlays/texture_name_relief__d_a.tga"`
  Defines the texture of blending with the surface of the terrain.
  For details on channels of `Texture` and `TextureRelief` textures, please see 3.1.2. Create necessary textures for an overlay section.
- `HeightMapBlendingContrast="0.9"`
  Spesifies how smooth the blending of the `HeightmapOffset` texture and the surface of the terrain will be.
- `WheelTracks="0"`
  Defines the transparency of tracks left on the overlay after truck's wheels. "0" corresponds to no tracks left.
- `IsExtrudable="false"`
  If this option is enabled and there is mud on the terrain under the overlay, the truck leaves ruts and throws clods of dirt. By default: "`true`".
- `GenerateDust="true"`
  If this option is enabled, the clouds of dust will appear on the overlay after physical interaction with the truck. This attribute is enabled for unpaved, dirt roads, for example.
- `IsAsphalt="true"`
  Sets the friction with the overlay surface corresponding to the friction with the paved road.
- `IsWetnessAsIce="true"`
  If the surface of the terrain under the overlay is brushed with Wetness, ice will appear on the overlay.
- `IsClampV="true"`
  Restricts the width of the road to the default width, so that it cannot be modified in the Editor.

Attributes for geometric overlays:

- `IsNotStaticSoil="true"`

If this attribute is disabled, clouds of dust and clods of dirt will appear when the track interacts with the overlay.

- `NoVeryFar="true"`
  Determines whether the overlay geometry will be hidden over long distances or not. Impacts the performance of the game.
- `BodyFriction="0.5"`
  This parameter specifies the friction of the overlay in case of collision. By default: “1”.

7.1.1. <Prebuild>

Describes the properties used by the Editor. However, when the Editor saves and rebuilds the map, these properties are packed into the level file, which is used by the game engine.

Attributes for texture overlays:

- `DefaultWidth="7"`
  Defines the default width of the overlay in meters.
- `FlattenPart="1"`
  Determines the power of flattening when the **Flatten** parameter in the Editor is enabled. The “1” correspond to absolute flattening, and the “0” – to no flattening.
- `HeightmapOffset=" texture_name__s.tga"`
  Is used to change the height of the terrain under the overlay in order to make a truck leave a rut.
- `SnowmapMask="texture_name__s_d.tga"`
  Is quite similar to `HeightmapOffset` parameter, but it corresponds to the ruts in snowy roads.
- `Layer="2"`
  Determines the priority of an overlay relative to another overlay if they are located on the same block of the terrain. The higher value implies the higher priority.
- `LandmarkUseAverageWidth="true"`
  If this option is enabled, the overlay's landmark on the minimap will be a constant average width, so that the actual widening or narrowing of the overlay will not be displayed on the minimap.
- `LandmarkColor="(140; 115; 90)"`
  Specifies the color of the landmark of the overlay on the minimap. The parameter is set in the RGB color format.
- `LandmarkPart="0.8"`
  Defines the part of the road wisth that will be shown on the mini-map. If the width of some fragment of the road was changed, it will be displayed accordingly on the mini-map.
- `LandmarkBorderWidth="1"`
  Determines the width of the landmark border displayed on the minimap.
- `LandmarkBorderColor="(80;80;80)"`

Specifies the border color of the landmark of the overlay on the minimap. The parameter is set in the RGB color format.

All parameters describing properties of landmarks are displayed both on the minimap in the Editor and in the game.

Attribute for geometric overlays:

- `OverlayType="Geometric"`
  This parameter is only used for geometric overlays, but not for texture ones. There are three main types of geometric overlays:
  - "**Geometric**" – overlays of this type don't hide their ends under the terrain and curve from point to point smoothly, like pipes;
  - "**GeometricConform**" overlays of this type hide their ends under the terrain and are pressed to the terrain, like borders;
  - "**GeometricWire**" - overlays of this type don't hide their ends under the terrain curve from point to point sharply at an angle, like wires.

7.2. <CombineXMesh>

The root tag of the XML file of the Mesh.

Attributes:

- `Type="Overlay"`
  The "**Overlay**" value specifies that this XML file of the mesh corresponds to an overlay.

7.2.1. <Material>

Similar to the `<Material>` tag used for models. See the 6.2.3. <Material> section fo more information.

When defining textures of this material (`AlbedoMap`, `NormalMap`, etc.) you need to specify paths to the textures that you put into the **Media\textures\overlays**. The paths here should be specified relative to the **Media\textures** folder.

8. Tags and Attributes of PbrMaterials

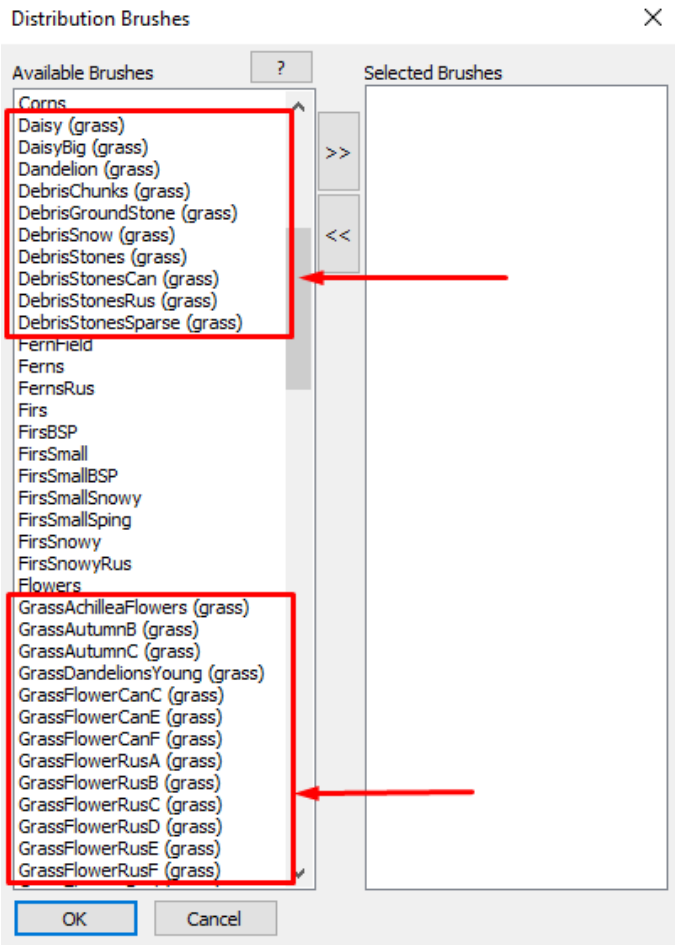
The section below provides some info on main tags and attributes used in XML descriptions of PbrMaterials.

8.1. <MaterialType>

The root tag of the XML file of the class of the PbrMaterial.

Attributes:

- `Texture="tiles/grass_rus_02__d_a.tga"`
  Defines the basic texture of the PbrMaterial.
- `Normal="tiles/grass_rus_02_relief__d_a.tga"`
  Defines the texture of blending with the surface of the terrain.
  For details on channels of `Texture` and `Normal` textures, please see 4.2. Create necessary textures for a PbrMaterial section.
- `GrassBrush="GrassShortRusA,GrassShortRus,GrassTallRusA"`
  Allows to add different types of grass to your PbrMaterial. You can look up names of grass types in the Distribution Brushes list in the Editor. All of them are marked with “(grass)” in this list.



- `AllowHiddenExtrudes="0.25"`
  Determines the maximum depth of extruded ruts.
- `IsThickGrass="false"`
  Defines whether the terrain grass under a PbrMaterial will be displayed or not.
- `Sound="surfaces/surf_grass_mix_loop"`
  Sets the main sound when the truck interacts with the surface. The general view of the value of this parameter is the following: `"surfaces/<sound_file_name>"`.

  NOTE: You can only use preset sound files for this parameter. The list of preset sound files names is performed in the Appendix A: IDs of Sounds Used for PbrMaterials.

- `EndEffectSound="surfaces/hit/hit_grass_rnd"`
  Sets the sound of effects when the truck interacts with the surface. For example, dust, pebbles, etc. The general view of the value of this parameter is the following: `"surfaces/hit/<sound_file_name>"`.

  NOTE: Only predefined sounds can be used here, see `Sound` above.

- `SkidSound="surfaces/skidding/skidding_grass_loop"`
  Sets the sound of interaction with the surface when the track is skidding. The general view of the value of this parameter is the following: `"surfaces/skidding/<sound_file_name>"`.

  NOTE: Only predefined sounds can be used here, see `Sound` above.

- `MinViscosity="2.0"`
  Determines the default coefficient of soil viscosity.
- `IsExtrudable="false"`
  Defines whether the ruts will be left on the terrain.
- `IsAsphalt="true"`
  Sets the friction with the overlay surface corresponding to the friction with the paved road. Defines whether mud traces will be left on the surface. If you paint the asphalt PbrMaterial with the Wetness brush, it becomes more slippery, but not softer.
- `WheelTracks="0"`
  Defines the transparency of tracks left on the overlay after truck's wheels. "0" corresponds to no tracks left.
- `NoGrass="true"`
  Defines whether any grass will be displayed on a PbrMaterial or not.
- `IsWetnessAsIce="true"`
  If the surface of the terrain under the PbrMaterial is brushed with Wetness, ice will appear on the surface.
- `IsSnowParticles="true"`

If this option is enabled, small particles of snow will fly out from under the wheels of a truck.

- `ShadingType="snow"`
  If this option is set as "snow", the PbrMaterial is considered as a snow layer. The essential parameter for snowy PbrMaterials creation.
- `ReliefNormals1="tiles/snow_forest_relief__d_a.tga"`
  Additional normal map for snowy layers. To get more info see 4.2. Create necessary textures for a PbrMaterial.
- `ReliefNormals1BlendContrast="0.7"`
  Spesifies how smooth the blending of the `ReliefNormals1` texture and the surface of the terrain will be.
- `ReliefNormals2="tiles/snow02_relief__d_a.tga"`
  Additional normal map for snowy layers. To get more info see 4.2. Create necessary textures for a PbrMaterial.
- `ReliefNormals2BlendContrast="0.3"`
  Same as `ReliefNormals1BlendContrast`.
- `ReliefNormals2BlendFactorAtNight="0.4"`
  Spesifies how smooth the blending of the `ReliefNormals2` texture and the surface of the terrain will be at night time.
- `SnowDetailNormal="tiles/snow_detail__n_d.tga"`
  Additional normal map for snowy layers. To get more info see 4.2. Create necessary textures for a PbrMaterial.
- `SnowDetailFactor="0.25"`
  Determines the power of applying `SnowDetailNormal` texture.
- `SnowDetailTiling="3.0"`
  Defines the size of visible particles of snow on the surface.
- `SnowTintBrightness="0.55"`
  Defines the brightness intense of a color when you paint the surface with the Colorization brush.
- `SnowAlbedoColor="(230; 230; 230)"`
  Defines the shade of white for snow layers.
- `UpVectorSnowAlbedoColor="(210; 210; 210)"`
  Sets the shade of snow put on houses, trees, rocks and other objects on snowy PbrMaterial.
- `WetnessAlbedoMultiplier="0.1"`
  Determines how much darker the shade of snow becomes when you paint the surface with the Wetness brush.
- `WetnessRoughnessMultiplier="0.1"`
  Determines how much glossier the snow becomes when you paint the surface with the Wetness brush.

9. Samples

NOTE: The main folder with samle materials is the following:
https://drive.google.com/drive/folders/1x4IF8c12-TQ_wiFlYoRh2QGb7VZvzCs8

9.1. Sample Model

All sample files used for creating a custom Model are located at the following archive:
https://drive.google.com/drive/u/0/folders/1EDfWMHFJ53jC4NnMQZi8NhBDeh69MaVw

9.2. Sample Overlays

All sample files used for creating a custom texture Overlay and a custom geometric Overlay are located at the following archive:
https://drive.google.com/drive/u/0/folders/1KcFkI5Un_-PHUZDRZAxQrDGQDs0rWLU4

9.3. Sample PbrMaterials

All sample files used for creating a custom PbrMaterial are located at the following archive:
https://drive.google.com/drive/folders/1Sr1RsgMXGxJxhVmhUvUFKZKhD_pLsHdJ

Appendix

Appendix A: IDs of Sounds Used for PbrMaterials

The table below contains names of predefined sound files that can be used for a custom PbrMaterial. You can set sound files for `Sound`, `EndEffectSound` and `SkidSound` parameters of the ***<MaterialType>*** tag. See the 8.1. <MaterialType> for more details.

№	*Sound file name*
`Sound` attribute of the ***<MaterialType>*** tag	
1	surf_asphalt_loop
2	surf_asphalt_rnd__1
3	surf_asphalt_rnd__2
4	surf_asphalt_rnd__3
5	surf_asphalt_rnd__4
6	surf_asphalt_rnd__5
7	surf_asphalt_rnd__6
8	surf_asphalt_rnd__7
9	surf_asphalt_rnd__8
10	surf_asphalt_rnd__9
11	surf_concrete_rnd__1
12	surf_concrete_rnd__2
13	surf_concrete_rnd__3
14	surf_concrete_rnd__4
15	surf_concrete_rnd__5
16	surf_dirty_loop
17	surf_dirty_rnd__1
18	surf_dirty_rnd__2
19	surf_dirty_rnd__3
20	surf_dirty_rnd__4
21	surf_dirty_rnd__5
22	surf_dirty_rnd__6
23	surf_grass_loop
24	surf_grass_mix_loop
25	surf_grass_mix_rnd__1
26	surf_grass_mix_rnd__2
27	surf_grass_mix_rnd__3
28	surf_grass_mix_rnd__4
29	surf_grass_rnd__1
30	surf_grass_rnd__2
31	surf_grass_rnd__3

32	surf_grass_rnd__4
33	surf_grass_rnd__5
34	surf_grass_short_rnd__1
35	surf_grass_short_rnd__2
36	surf_grass_short_rnd__3
37	surf_grass_short_rnd__4
38	surf_grass_short_rnd__5
39	surf_grass_short_rnd__6
40	surf_grass_short_rnd__7
41	surf_grass_short_rnd__8
42	surf_grass_short_rnd__9
43	surf_grass_short_rnd__10
44	surf_grass_short_rnd__11
45	surf_grass_short_rnd__12
46	surf_grass_short_rnd__13
47	surf_grass_short_rnd__14
48	surf_grass_short_rnd__15
49	surf_grass_small_loop
50	surf_grass_tall_rnd__1
51	surf_grass_tall_rnd__2
52	surf_grass_tall_rnd__3
53	surf_grass_tall_rnd__4
54	surf_grass_tall_rnd__5
55	surf_grass_tall_rnd__6
56	surf_grass_tall_rnd__7
57	surf_grass_tall_rnd__8
58	surf_gravel_loop
59	surf_gravel_rnd__1
60	surf_gravel_rnd__2
61	surf_gravel_rnd__3
62	surf_gravel_rnd__4
63	surf_gravel_rnd__5
64	surf_gravel_rnd__6
65	surf_ground_loop
66	surf_ground_rnd__1
67	surf_ground_rnd__2
68	surf_ground_rnd__3
69	surf_ground_rnd__4
70	surf_ground_rnd__5
71	surf_ground_rnd__6
72	surf_ground_rnd__7
73	surf_ground_rnd__8
74	surf_ground_rnd__9

75	surf_ice_break_rnd__1
76	surf_ice_break_rnd__2
77	surf_ice_break_rnd__3
78	surf_ice_break_rnd__4
79	surf_ice_break_rnd__5
80	surf_ice_break_rnd__6
81	surf_ice_break_rnd__7
82	surf_ice_break_rnd__8
83	surf_ice_loop
84	surf_ice_rnd__1
85	surf_ice_rnd__2
86	surf_ice_rnd__3
87	surf_ice_rnd__4
88	surf_ice_rnd__5
89	surf_ice_rnd__6
90	surf_rock_loop
91	surf_rock_rnd__1
92	surf_rock_rnd__2
93	surf_rock_rnd__3
94	surf_rock_rnd__4
95	surf_rock_rnd__5
96	surf_sand_loop
97	surf_sand_rnd__1
98	surf_sand_rnd__2
99	surf_sand_rnd__3
100	surf_sand_rnd__4
101	surf_sand_rnd__5
102	surf_sand_rnd__6
103	surf_snow_rnd__1
104	surf_snow_rnd__2
105	surf_snow_rnd__3
106	surf_snow_rnd__4
107	surf_snow_rnd__5
108	surf_snow_rnd__6
109	surf_snow_rnd__7
110	surf_water_eq_loop
111	surf_water_loop
`EndEffectSound` attribute of the ***<MaterialType>*** tag	
1	hit_grass_rnd__1
2	hit_grass_rnd__2
3	hit_grass_rnd__3
4	hit_grass_rnd__4
5	hit_grass_rnd__5

6	hit_grass_rnd__6
7	hit_grass_rnd__7
8	hit_grass_rnd__8
9	hit_grass_rnd__9
10	hit_grass_rnd__10
11	hit_gravel_rnd__1
12	hit_gravel_rnd__2
13	hit_gravel_rnd__3
14	hit_gravel_rnd__4
15	hit_gravel_rnd__5
16	hit_gravel_rnd__6
17	hit_gravel_rnd__7
18	hit_gravel_rnd__8
19	hit_gravel_rnd__9
20	hit_gravel_rnd__10
21	hit_mud_rnd__1
22	hit_mud_rnd__2
23	hit_mud_rnd__3
24	hit_mud_rnd__4
25	hit_mud_rnd__5
26	hit_mud_rnd__6
27	hit_mud_rnd__7
28	hit_mud_rnd__8
29	hit_mud_rnd__9
30	hit_mud_rnd__10
`SkidSound` attribute of the ***<MaterialType>*** tag	
1	skidding_concrete_loop
2	skidding_dirty_loop
3	skidding_grass_loop
4	skidding_gravel_loop
5	skidding_ground_loop
6	skidding_ice_loop
7	skidding_snow_loop