

The Pipeline

The pipeline is a set of actions, embedded in a flow line, specified by one or several XML files, which generates an image according to the data input entered. The actions are headed to define the shader, set the data input to the shader, assing a rendering context and to process the resulting output.

Using the Pipeline

The Pipeline is a middle point between the data and the algorithms we have to represent data. Using the pipeline we can configure the way the data is processed, which shader should use it and how to handle the output (store it in a texture for example). It allows the programmer to easily prototype render algorithms by just editing an XML file. Thanks to this class we can share render techniques between different applications without having to recompile the source code.

Inside the Pipeline we configure the actions needed to render the object, redirecting the input to the proper shader variables, or making several passes.

The render flow defined by the Pipeline could be executed as many times as we need, but the pipeline has its own way to handle the whole render of the scene. It means it could have steps that checks pipeline's state or which inputs we already have and do different things according to that, bringing totally freedom to prototype whatever we need.

Check the [Pipeline?](#) to see the complete actions list.

Check the [Pipeline XML Example?](#).

Basic actions list

Here is a list of all the actions supported by the Pipeline. The name is the same as the XML tag that represents it, and the parameters are the attributes of the XML tag.

- **code**: It encapsulates a serie of actions. Being encapsulated makes easier to repeat the same fragment of code several times during the execution.
- **shader**: A shader tag represent a vertex shader and a pixel shader. It is important to define the set of macros inside this tag to configure the shader. The tags can be conditioned to the pipeline's state so depending on which inputs we have the shader can act different.
- **constant**: It contains a variable, it can come from the application (then we just redirect it to the shader) or can be defined inside the tag. If it is defined in the action then the type must be specified.
 - ◆ **name**: the var name. If there is no target specified the target will be the one with the same var.
 - ◆ **type**: data type, it could be "bool", "float", "int",
 - ◆ **value**: the value in text format ("1.0,1.0,1.0", "true", etc)
- **buffer**: a buffer contains a stream of data assigned per vertex. It is redirected to the shader, if there is no target attribute then it is put it in the stream with the same name inside the shader.
- **sampler**: redirects a texture from the input to the shader. Works similar to a buffer but it can be specified a texture from the hard drive directly
 - ◆ **filename**: using the filename we can put a texture fro mthe hard drive directly in the shader.
- **camera**: it allows to get information regarding to the camera, like the projection matrix, modelview, eye pos, etc.

- **condition:** allows to split the flow according to a set of conditions, for further information check the conditions chapter in this page.
- **draw:** tells the hardware to start rendering.
- **target:** allows to render the scene into a texture of a given size.

Shader

The pipeline needs at least one shader to execute the algorithms. All the steps are headed to provide the appropriate input to all the shaders involved in the render. The shaders are compiled on demand, this means there is not only one shader in memory per pipeline, instead of that, they are compiled according to the set of macros needed and stored for further use. This way to handle the shaders have a significant overhead in the performance, that's because the programmer has the possibility to give an identifier to the input, using an id we can store precompiled shaders and retrieve them according to that id. Using the id we skip all the defining steps.

```
<shader vs="data/shaders/global.vsh" ps="data/shaders/global.psh">
  <define name="USE_NORMALS"/>
  <define name="USE_SPECULAR" if_input="light0_pos"/>
  <define name="USE_SKINNING" if_input="bones"/>
  <define name="USE_DIFFUSE_TEXTURE" if_input="diffuse_texture"/>
</shader>
```

Redirecting data

Most of the actions in the Pipeline are headed to redirect the input we have to different places in the shaders. There are different situations according to what we want:

- The input gets blocked in the pipeline. It means we don't want to use an input we have in the pipeline. Then we just skip the action in the pipeline.
- The input pass through. In that situation the input is placed in the shader's var with the same name. We just need to say the name of the input and nothing else.
- The input is redirected. In that situation the input is placed in a var with a different name than the input. To do that we need to specify the name and the target.

Conditions

Every action can have a different set of conditions, only if all the conditions are true the action will be executed. This is useful when you want to reuse the same pipeline with different sets of inputs (like a mesh without textures, or an object without lighting).

To do that we can add the condition attributes in the specific action, the supported conditions are:

- **if_input:** if there is a defined input using that name.
- **if_not_input:** if there is no input defined with that name.
- **if_shader:** if the shader currently in use has this macro defined
- **if_not_shader:** if the shader currently in use doesn't have this macro defined
- **if_name:** if the object's name being rendered is equal to that
- **if_not_name:** if the object's name being rendered is not equal to that
- **if_key:** if the keyboard key is pressed
- **if_not_key:** if the keyboard key is not pressed
- **if_tag:** if the tag is set in any level of the parameter container stack

- **if_not_tag**: if the tag is not set in any level of the parameter container stack
- **if_top tag**: if the tag is set in the top level of the stack
- **if_not_top tag**: if the tag is not set in the top level of the stack
- **if_eval**: if the result of evaluate the comparison is true
- **if_infrustum**: if the AABB of the item in the top of the stack is inside the frustum of the active camera.

Example of condition embedded in a regular action:

```
<buffer name="normal" if_input="normal"/>
```

XML doesn't allow to have several attributes with the same name, so if you want to use AND conditions you should put the values separated by commas, and if you want to use an OR condition then separate the values by '|'.

The next example will execute the code "shadowmap" only if there is one of the two tags (SPOT_LIGHT or OMNI_LIGHT) and if the shader has both keys defined (USE_LIGHT and CAST_SHADOWS). {{{#!cplusplus <execute name="render" if_tag="SPOT|OMNI" if_shader="USE_LIGHT,CAST_SHADOWS" /> }}}}

And if what we want is to create a set of actions executed under a condition with its proper else, then we use the **condition** action.

Example of conditional action:

```
<condition if_shader="USE_LIGHT">
  <constant name="ambient_color" type="vector3" value="0,0,0" />
  <else>
    <constant name="ambient_color" type="vector3" value="1,1,1" />
  </else>
</condition>
```

State

If we pretend to use several passes in the render then we need a proper way to blend all different renders. That is why the State action is so important. It changes the way the 3d API renders the pixels, like testing occlusion or mixing pixels with the previous ones.

Here is a list of all the properties we can change of the render state:

- **COLOR_MASK**: Sets the range of colors to be used.
- **DEPTH_TEST**: To switch the occlusion test, it is a boolean value.
- **DEPTH_WRITE**: It changes if the depth buffer is updated or not. It is a boolean value.
- **DEPTH_FUNC**: It controls the function used to determine if a pixel is occluded or not, possible values are NEVER, EQUAL, LEQUAL, GEQUAL, GREATER, LESS, NOTEQUAL, ALWAYS.
- **ALPHA_TEST**: It enables the alpha testing.
- **BLEND**: Enables the blending between new pixels and the ones already in the buffer.
- **BLEND_FUNC**: Changes the blending function for the written pixel and the previous pixel, possible values are ZERO, ONE, DST_COLOR, ONE_MINUS_DST_COLOR, SRC_ALPHA, ONE_MINUS_SRC_ALPHA, DST_ALPHA, ONE_MINUS_DST_ALPHA, CONSTANT_COLOR_EXT, ONE_MINUS_CONSTANT_COLOR_EXT, CONSTANT_ALPHA_EXT, ONE_MINUS_CONSTANT_ALPHA_EXT, SRC_ALPHA_SATURATE.
- **FILL_MODE**: Allows the filling mode for every triangle (SOLID or WIRE)
- **CULL_MODE**: Enables the cull face feature (FRONT, BACK, FRONT_AND_BACK, NONE)

- **CLEAR:** Clear the buffers.
- **CLEAR_COLOR:** Sets the clear color.
- **CLEAR_DEPTH:** Sets the depth clearing color.
- **POLYGON_OFFSET:** Sets both values for polygon offset.

Setting up the input through the ParameterContainer?

Once we have designed our render Pipeline we can attach it to our program. We need to add the set of data that the pipeline will use, like the streams with all the vertices and normals, the textures, constants and so.

To do that we need a container where all the data can be dumped in a way the pipeline can use it. This is the ParameterContainer?.

Inside a ParameterContainer? we have different parameters that represent data, every parameter has a string defining its name and its key. If we want a parameter we just ask the ParameterContainer?.

To fill our parameter container we just call the 'add' method:

```
ParameterContainer* container = new ParameterContainer;
container->add("camera", mycamera); //allows to get camera info from the pipeline
container->add("ambient_color", scene->getAmbientColor() ); //assign a vector
container->addFromMesh(mesh); //sets all the streams from a mesh
container->add("specularmap_texture", my_texture); //sets a texture
```

How to execute the pipeline

Once the ParameterContainer? is full with all our data we give it to the Pipeline and call for execute:

```
pipeline->setContainer( container );
pipeline->execute();
```

Using it properly

Although we have seen all the features regarding to the pipeline executing workflow, there are some points we need to take into account.

- The pipeline is not meant to be used in applications with thousands of instances because of the performance's overhead. It is very fast and useful to prototype and test algorithms before they are integrated into a final application. It is very useful as an extension of the capabilities of the applications in terms of rendering. At last, it is extremely useful in applications where the rendering of the meshes are composed of multiple, complex and heterogenous rendering passes, that is the case of the photorealistic rendering of characters and environments.
- The more actions and conditions it have the more time takes to have the object rendered.
- The overhead in the use of the Pipeline comes from the number of steps inside, but not from the complexity of the input. It means the overhead is not a problem for a small amount of objects even if we have a big set of data as input.