Dent Documentation

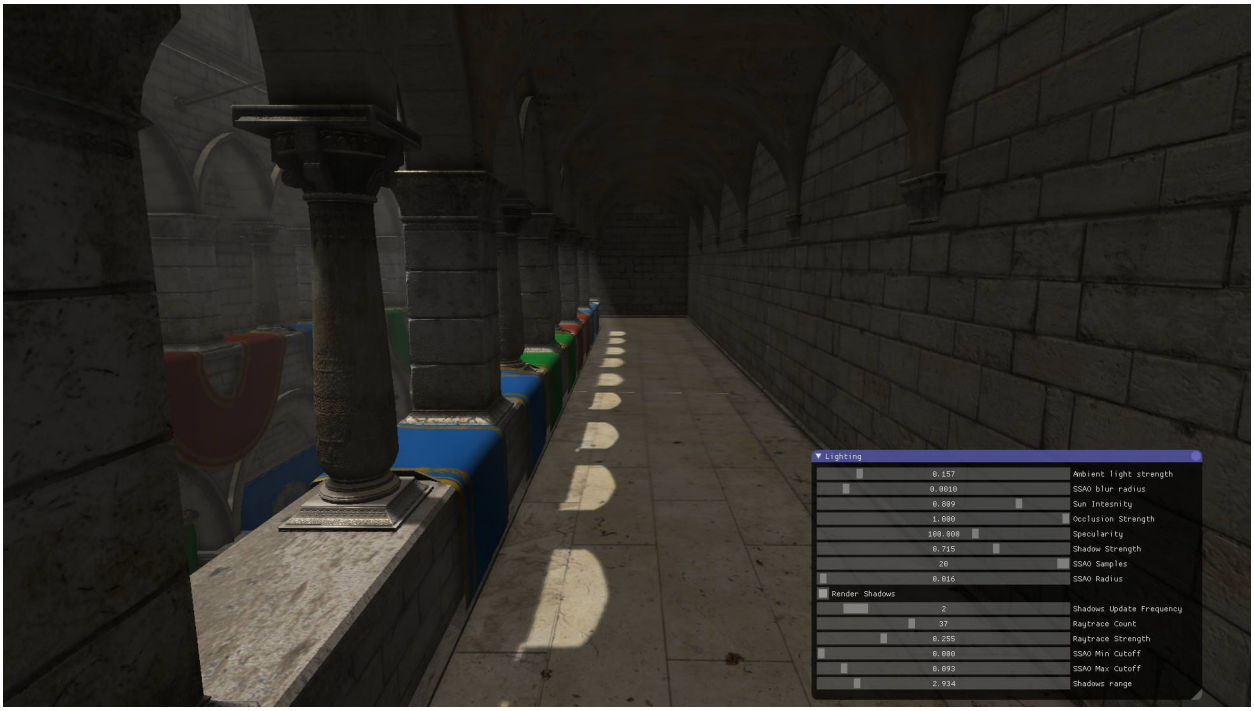
Release 0.1

Robert Spencer

Dec 07, 2019

Contents

Dent is a simple OpenGL 4 game engine written in Python. Dent has support for asset importing, shader management and event handling, as well as a plug-and-play deferred rendering pipeline.



There is no editor for Dent projects (yet): everything is simply pure Python files (and assets etc.) as it is meant to be.

The best way to learn about it is by examples and tutorials.

CHAPTER 1

Render Pipelines

1.1 Foreword

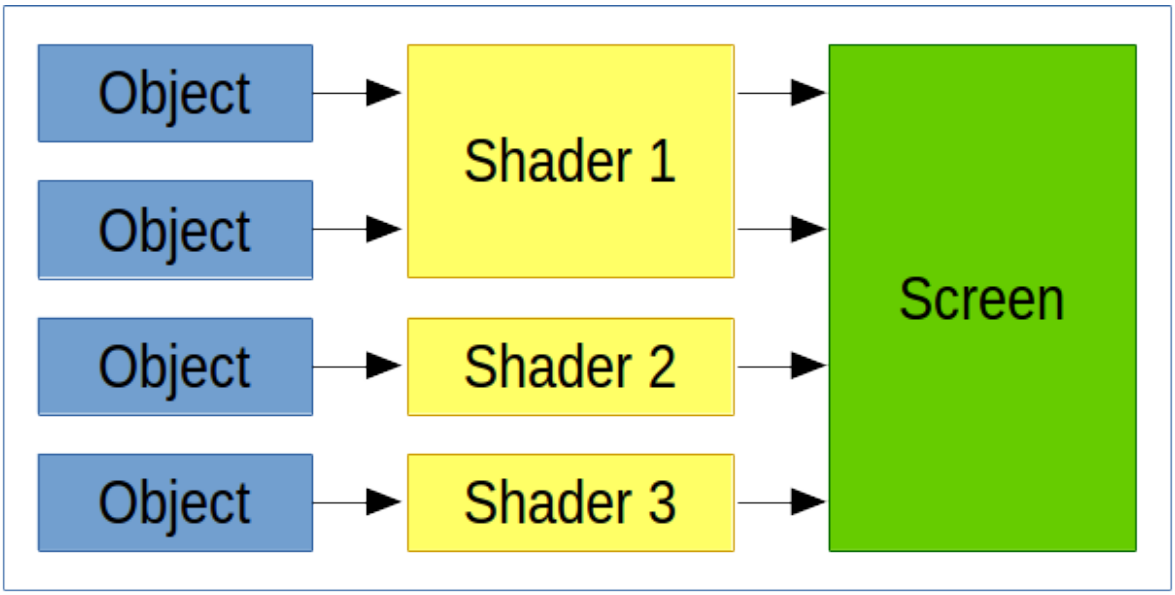
Working with render pipelines can be complicated. As such it is the job of the engine to make that complication go away. If you are not interested in the behind-the-scenes information about render pipelines, skip to the last section to find out why you don't need to know about them.

1.2 Overview

To display anything on the screen, it must go through a render pipeline. Dent affords a large degree of control over this pipeline to the programmer, while simultaniously making "standard" pipelines quick to set up.

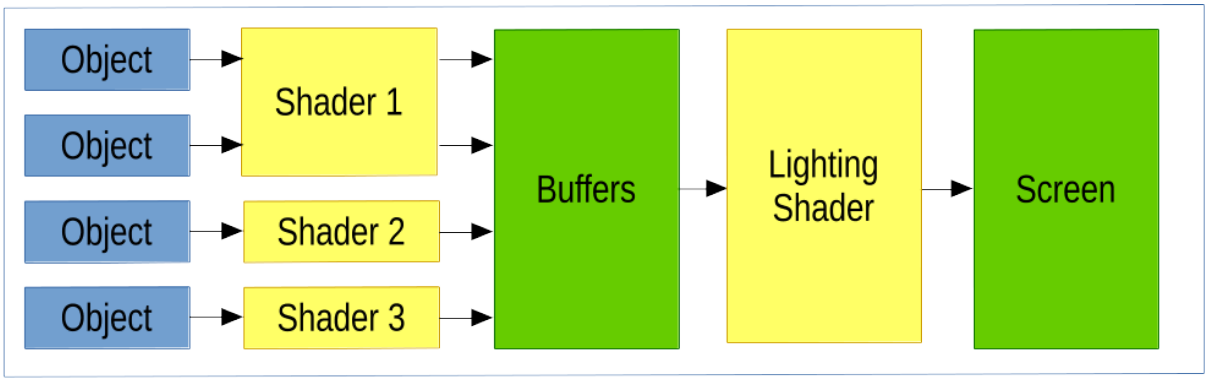
The two main render strategies are deferred rendering and forward rendering, and we will use these as examples, although other, more complicated strategies are possible. This is not a complete guide. For a more complete story see here.

Forward rendering is the simpler to think about and is probably better for starting out. In this model, geometry is drawn directly to the screen.



Note that each object is fully drawn one at a time. There may be different shaders for different objects (eg different lighting effects for water or for glass), and multiple objects may be drawn by the same shader.

In deferred rendering, the output is post processed after everything has been drawn. Then a shader is applied to the entire scene simultaniously.



The buffers are diffuse, normal and position maps of the entire scene, and the first shaders are responsible for populating those maps.

A set of such buffers is a `RenderStage` and the whole pipeline is managed by a `RenderPipeline` object. A pipeline has a number of stages, typically culminating in the screen buffer. When `run`, it will load each of the stages in sequence and execute some drawing code for each.

Thus in our direct rendering above, we might have a render pipeline:

```
RenderPipeline(
  [
    RenderStage(render_func=display_function, final_stage=True)
  ]
)
```

While in the deferred rendering, we might have:

```
RenderPipeline(
  [
    RenderStage(render_func=display_function, aux_buffer=True),
    RenderStage(render_func=lighting_function, final_stage=True)
  ]
)
```

All of the calls to *glDraw** happen in the *display_function* and the *lighting_function*. However, these too will be abstracted away by dent so that, for example, the lighting display function might just be:

```
def lighting_function():
  lightingRectangle.display()
```

where the lighting rectangle abstracts away the geometry and shader needed to do the per fragment lighting calculations.

Render pipelines are very simple (take a look at the source code) and as such, can be made to be very powerful. The can have multiple stages for multiple lighting/postprocessing passes or render to textures to be used elsewhere in the game.

1.3 Pipelines in Scenes

A Dent `Scene` has one render pipeline to render the scene to the display. To help set this up (although you are welcome to do it manually), the standard scene comes default with a `renderPipeline` attribute set up as per the first example above. The display function for this pipeline is the `display` function of the scene. Thus, sufficient bootstrap code is:

```
class MainScene(Scene):
  def __init__(self):
    super(MainScene, self).__init__()
    # Make objects here

  def display(self, **kwargs):
    for object in self.objects:
      object.display()
```

For a deferred rendering scene, extend `DeferredRenderScene`. Again, the display function is `display`.

CHAPTER 2

Tutorial: A Shadertoy-like Program

Shadertoy is a website dedicated to shader-only programs. It is well worth checking out what you can do with just a fragment shader.

Let us write a fragment shader viewer in Dent. We'll go about it in four short stages.

2.1 Stage 0: Planning

Before diving in, lets plan what we need.

We want to just run a fragment shader. But to render that we need some geometry on which to execute it. We can just use a rectangle that fills up the whole screen (Dent has us covered here so we don't have to mess with any vertex data etc.).

We'll have a vertex shader that is as simple as can be, and then we can do magic in the fragment shader.

2.2 Stage 1: Getting a Game Running

We can use the dent script `dent-init` to set up our game. Simply run:

```
$ dent-init tutorial
```

in order to make a minimal game tree.

Lets take a look at what's been done. We now see a directory called `tutorial` under which all our code and assets etc will go.

The entrypoint to our game (ie the file we'll run) is called `tutorial`. We don't need to do any setup before the game runs, so this file just has a single import statement. Dent is unusual (underdeveloped) in that the game begins when you import that module. This is earmarked to be changed in the near future.

Every game needs a scene, and we have a dummy one, `scenes/MainScene.py`:

```
from dent.Scene import Scene

class MainScene(Scene):
  pass
```

The file `scenes/__init__.py` does some python and dent housekeeping. It is unimportant at the moment

Running the `tutorial` file should present you with a blank scene. Mmmm. Progress!

2.3 Stage 2: Writing a Shader

To go much further we are going to need a shader. You can look at the *Shaders* docs in order to get more detail as to how shaders are handled in Dent, but for now we will need a *vertex* shader and a *fragment* shader. The fragment shader will eventually hold all your ray-tracing/edge-detecting/awesome drawing code, but to start with can be very simple.

Make a new directory, `shaders/main` and in it add `fragment.shd`:

```
#version 400
out vec4 fragColor;
in vec2 pos;

void main()
{
  fragColor = vec4(
    (pos.x + 1) / 2,
    (pos.y + 1) / 2,
    cos(pos.y * 3 + pos.x * 8) / 2 + 1,
    1);
}
```

Here we are expecting an input of `pos` from the vertex shader and will give an output called `fragColor`. We are just setting the color to be a set of diagonal stripes (as we shall see).

Then create the vertex shader, `vertex.shd`. The vertex shader will be getting input from Dent, so its input needs to be appropriately named. Dent can provide shaders will all manner of inputs: positions, normals, model matrices and more. However, for this tutorial, we only need the position of the geometry:

```
#version 400
in vec3 position;
out vec2 pos;

void main()
{
  gl_Position = vec4(position, 1);
  pos = position.xy;
}
```

Here we have assumed that the position passed to the shader is in screen space: that is, it needs no model matrix nor view matrix. This will turn out to be a good assumption.

Note, if you are unfamiliar with GLSL, that *pos* is a variable passed from the vertex shader to the fragment shader and *position* is a variable passed to the GPU from the geometry of the object being rendered.

2.4 Stage 3: Using the Shader

To actually use the shader we need geometry. We want a quad that will fill up the entire screen. Thus we will need four points at positions `(-1, -1)`, `(-1, 1)`, `(1, 1)` and `(1, -1)`. We will also want two triangles. If you know about geometry culling, then you will know that these triangles will need to "face" the right way. We'll then need to send this geometry to the GPU and bind the correct vertex attributes to the shader.

Or rather, Dent will have to do all of that. Because rendering rectangles is a common task (think minimaps, icons and text), and that is what engines are for, we will delegate that task to it.

Dent defines a `RectangleObject` that does just this. By default it creates a square from `(-1, -1)` to `(1, 1)` just as we need. When constructing it, we pass it the name of the shader that it should use to render. Since we put our earlier shader in `shaders/main`, this will be the string "main".

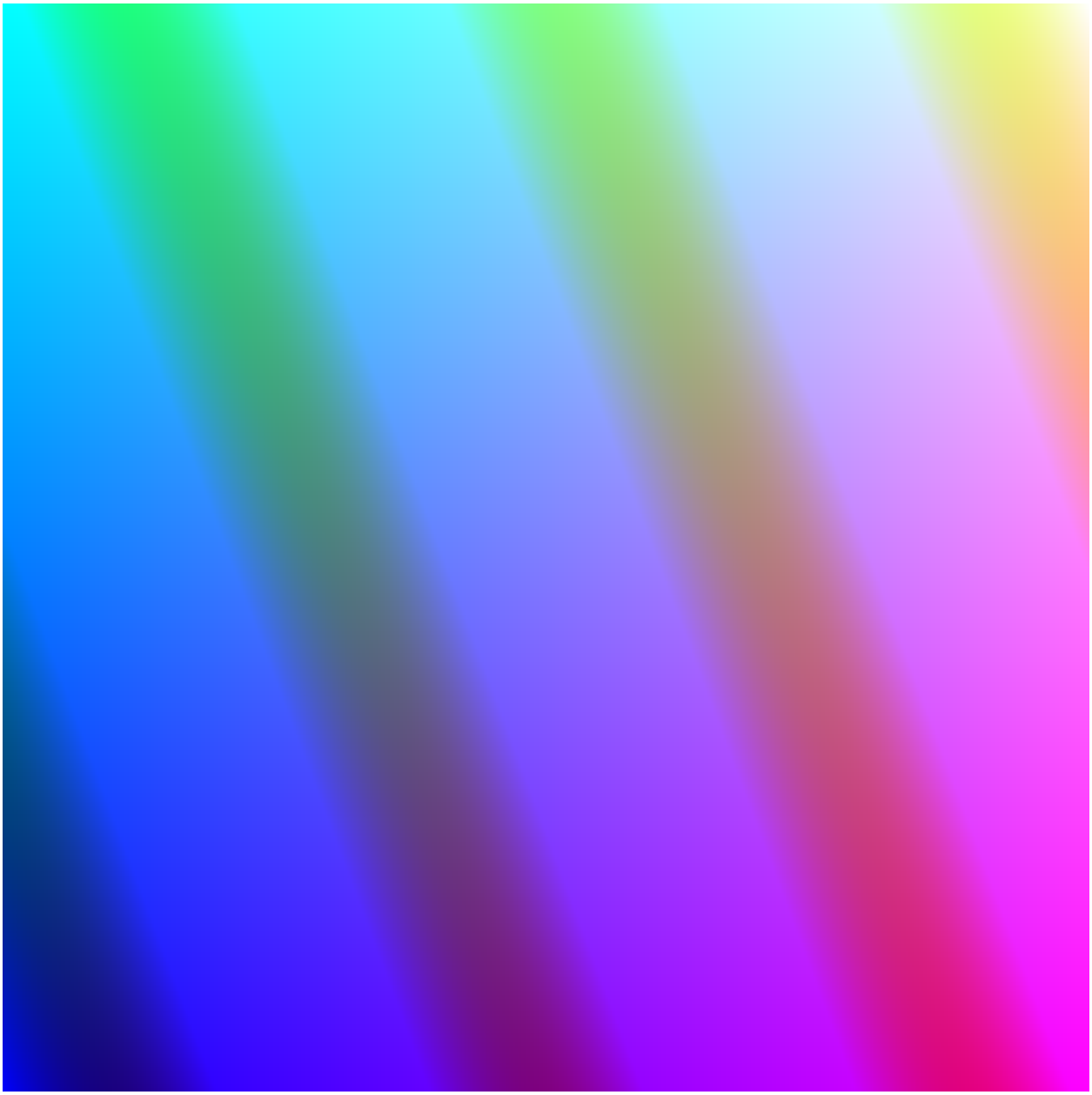
Hence the change to the scene is quite minimal. We also define a `display` function that is called to refresh the screen:

```
from dent.RectangleObjects import RectangleObject

class MainScene(Scene):
  def __init__(self):
    ...
    self.object = RectangleObject('main')

  def display(self, **kwargs):
    self.object.display()
```

All things being well, running the game should give us:



2.5 Stage the last: Afterword

At this point we can leave the Python (and Dent) code and focus on our shader. As mentioned before, there is a lot one can do with just a fragment shader. For ideas, check out the Shadertoy gallery.

However, at some point you will probably want to give your shader an image to work with. In GLSL, this is called a `sampler`. Using Dent to do this is a little rough at the moment, but simply add the following to the `__init__` function of your scene:

```
def __init__(self):
  ...
  self.texture = dent.Texture.Texture(dent.Texture.COLORMAP)
```

(continues on next page)

(continued from previous page)

```
  self.object.shader['colormap'] = dent.Texture.COLORMAP_NUM
  self.texture.loadFromImage('imagefile.png')
  self.texture.load()
```

Then you can use the texture in the shader as a `uniform sampler2D` called `colormap`.

CHAPTER 3

Shaders

Shader objects in Dent represent complete shader programs: a vertex shader, possibly tesselation shaders and geometry shader, and a fragment shader. They are written in OpenGL Shader Language, with a small modification.

This guide assumes some knowledge of OpenGL shaders and pipelines. If you are new to all this, it is suggested to copy one of the shaders from an existing project and tweak it to your needs, or only use the builtin shaders.

Shader type	Filename	Required
Vertex	`vertex.shd`	Yes
Geometry	`geometry.shd`	No
Tesselation control	`tesscontrol.shd`	No
Tesselation evaluation	`tesseval.shd`	Only if control shader present
Fragment	`fragment.shd`	Yes

A simple vertex shader might look like this:

```
#version 400
in vec3 position;
out vec2 pos;

uniform mat3 model;

void main()
{
  gl_Position = vec4((model * position.xyz), 1);
  pos = position.xy/2+0.5;
}
```

The corresponding fragment shader might be:

```
#version 400
in vec2 pos;
out vec4 fragColor;
```

(continues on next page)

(continued from previous page)

```
uniform sampler2D colormap;

void main()
{
  fragColor = texture(colormap, vec2(pos.x, 1-pos.y));
}
```

Note the output of the fragment shader is a *vec4*. You may output up to three vectors for defered rendering (see *Render Pipelines*).

Shaders are stored in the game tree under the folder `shaders`:

```
game
  ├── my-awesome-game.py
  ├── scenes
  │   ├── __init__.py
  │   └── MainScene.py
  └── shaders
      ├── image
      │   ├── fragment.shd
      │   └── vertex.shd
      └── ...
```

A shader object is created easily. For example to create a standard vertex-fragment shader and set some uniforms:

```
import dent.Shaders

...

shader = dent.Shaders.getShader('images')
shader['some_uniform'] = 1.4
shader['some_other_uniform'] = np.arange(1, 4, 0.3)
```

This corresponds to the `shaders/image/*` shader above.

The main function of shaders is the `draw` method. This loads the shader, sets the relevant uniforms and executes a `glDraw*` command. The precice command depends on the type of shader (generic, instanced, or feedback). Thus an object in the scene typically has a `display` function of the form:

```
def display(self):
  self.shader['model'] = self.model
  self.shader.draw(gl.GL_TRIANGLES, self.renderID)
```

For more detail, see the API documentation of the api/dent.Shaders.

CHAPTER 4

Actions and Animations

Any `Object` may have a `ActionController` assigned to it. If so, this action controller will take control if the object's `position` and `angle`.

You add animations to an action controller through the object, calling `add_animation()`. The action controller will at all times select an animation and apply it to the object.

Currently animations *must* be humanoid with a `Hips` bone at the root, as the motion of this bone will be used as the basis of the motion of the object itself.

When an action completes, the action controller will select a new action to apply to the object. This will be chosen according to some *weights*. By default the weights of the actions are all equal, but you can override this by supplying a `action_weight` function.

CHAPTER 5

Messaging

Dent runs its events off a messaging system. You can hook into this in order to access system events such as keyboard input, mouse input and timers. Even more, you can insert messages into the queue. This is the recommended way to communicate between elements of your game.

Dent also saves a log of the message queue every time your game is run.

If you use messaging for all non-deterministic interactions in your game (random die rolls for example), then Dent will be able to read a message log as a game replay. This is useful for debugging.

To fire an event, call:

```
message = dent.messaging.Message('event_name', ('some', 'data'))
dent.messaging.add_message(message)
```

Here we have made an event of type *event_name* and with two data.

To add a handler, simply call:

```
dent.messaging.add_handler('event_name', handler_func)
```

The hander function must expect exactly the data that will be in the event. Thus, in the above case it must take two parameters. The handler will be called with `some` and `data` as the parameters, given the previous message.

Naturally there can be mulitple handlers for the same event from different parts of the system. Thus, for example, a weapon object and the camera object might define hooks for keyboard input.

5.1 Default Messages

Dent fires off a number of messages that you may write hooks for. They are

Message Type	Description	Arguments
`mouse`	A mouse button has been clicked	The mouse button, state and xy coordinates
`mouse_motion`	The mouse has moved	The xy coordinates of the mouse
`keyboard`	A key has been pressed	The pressed key character
`keyboard_up`	A key has been released	The released key character
`timer`	Fired every timer tick	The current number of frames per second
`game_start`	Fired once at the beginning of the game	None

5.2 API

class dent.messaging.**Message**(*message_type*, *data=()*)
Bases: object

dent.messaging.**add_handler**(*message_type*, *handler*)

dent.messaging.**add_message**(*message*)
Adds a new message to the queue.

dent.messaging.**game_start_handler**(*time*)

dent.messaging.**load_messages**(*filename='replay.log'*)
Loads all messages in a given file.

dent.messaging.**load_replay**(*filename='replay.log'*)
Must be called when the only item on the message queue is *begin_game*. Loads the message queue from the given file.

dent.messaging.**process_messages**()

dent.messaging.**save_messages**(*filename='replay.log'*)
Saves all processed and unprocessed messages to a file.

CHAPTER 6

Materials

Materials are nothing more than a glorified set of parameters to the rendering software (shader). In theory, the rendering system is simply:

```
render(mesh, material)
```

where the mesh specifies the geometry, and the material everything else.

In practice, there are a number of different rendering systems and even more model formats that store materials, and as such, the `Material` class must cater to all of them. Dent tries to take a "catch-em-all" approach, where each material has as many parameters as it can, and it is up to the renderer to discard that are not useful.

6.1 Parameters of a Material

If you are writing a shader for the default Dent material system, you can expec to be fed a number of textures and constants. They are as listed at the bottom of this document, with details. In brief, though, they are:

parameter	uniform name	type	description
diffuse color tint	`diffuse_tint`	`vec3`	a constant by which to multiply the colormap
metallic tint	`metallic_tint`	`float`	a constant by which to multiply the metallic map
roughness tint	`specular_tint`	`float`	a constant by which to multiply the roughness map
color texture	`colormap`	`sampler2d`	the fragment diffuse/albedo value
normal texture	`normalmap`	`sampler2d`	the fragment normal map
specularity texture	`specularmap`	`sampler2d`	the fragment specularity value
metallic texture	`metallicmap`	`sampler2d`	the fragment metallic value
roughness texture	`roughnessmap`	`sampler2d`	the fragment roughness value

As you can see, this covers most of the required inputs to Blinn-Phong or Cook-Torrance BDRFs, and so can be used with the builtin lighting shaders.

6.2 Parsing Materials from Models

Importing materials is a pain for two reasons. The first is that, unlike meshes, there's no consensus on what information should be stored with an object with regard to its materials. The second is the way that `pyassimp` deals with this problem: it doesn't.

As a result, under the hood, Dent has to do some fancy footwork to guess the right parameters for the material. This unfortunately may require some work from the artist/developer. The exact specifics of the defaults chosen are detailed below.

6.3 Parameters

All texture maps described below can be overridden (see *Overriding Textures*).

6.3.1 Diffuse and Albedo

Most shaders require some sort of colour data. For Phong shaders, this is the diffuse colour. For PBR it is known as the albedo.

In Dent this data is passed as the diffuse colour tint and the colour texture. The correct way to compute the actual value is to multiply the two together.

The default for the image is a blank white texture. The default for the colour is the assimp default of all black. The tint can be specified in any way that is parsed to the `COLOR_DIFFUSE` property of assimp, and the map in any way that can be parsed to the first texture in the diffuse stack. As an example, in `OBJ` material format:

```
newmtl my_material
Kd 0.640000 0.640000 0.640000
map_Kd color.tga
```

6.3.2 Normal Maps

Normal maps are also stored as part of materials. They are only stored as textures.

The default texture is a single `(0,0,1)` valued `(r,g,b)` texture. Any map passed to the first layer of the normal stack of assimp is valid, as is any map passed to the first layer of the bump map. As an example in `OBJ` material format:

```
newmtl my_material
map_Bump normal.tga.
```

6.3.3 Phong Specific Parameters

The specularity is a measure of the coefficient of specularity in Blinn-Phong lighting. It is stored as a texture only. Any map passed to the first layer of the specular stack of assimp is valid. As an example in `OBJ` material format:

```
newmtl my_material
map_Ks specular.tga
```

6.3.4 PBR Specific Parameters

The parameters of roughness and metallic are specific to PBR shading. They are stored in the roughness texture and tint, and the metallic texture and tint parameters. Again, the correct value is obtained by multiplying the two.

Currenly the only way to set these textures is to override them. The tints default to 1.

6.4 Overriding Textures

Any of the above textures can be overridden, by placing appropriate image files in the same directory as the model. The format for the image filename is `{material_name}.{suffix}.png`, where the `{suffix}` is given from the below table

Texture type	Suffix
Diffuse	`diff`
Normal	`norm`
Specular	`spec`
Roughness	`roug`
Metallic	`meta`

6.5 Debugging Materials

You can use the Dent asset inspector to view materials and their properties, once they have been loaded into an asset datastore. This is useful to study the values for the tints and textures that are actually parsed by Dent. To do this, run `dent-assets inspect`, and navigate to the material in question and hit enter to examine its properties.

6.6 API

CHAPTER 7

Textures

7.1 API

Python Module Index

d

Index