Middle East Technical University Department of Computer Engineering

CENG 795

Advanced Ray Tracing

Fall ’2022-2023
Assignment 4 - All About Textures
(v.1.0)

Due date: December 15, 2022, Thursday, 23:59

1 Objectives

In this homework you are expected to add texture mapping capabilities to your ray tracers. We will be using textures for five main purposes:

- Background textures for rays not hitting any objects (replace_background)
- Diffuse and specular reflectance maps (replace_kd, blend_kd, replace_ks) for objects
- To disable shading and simply paste the texture color on an object (replace_all)
- Normals maps (replace_normal)
- Bump maps (bump_normal)

We will be dealing with two types of textures:

- Image textures (typically PNG and JPEG files – you can use any image reader library)
- Perlin noise (generated for each point on the fly)

Keywords: *multisampling, aliasing, reconstruction, filtering, distribution ray tracing*

2 Specifications

1. The specifications are the same as for the previous homeworks so are not repeated here.

3 Scene File

Please see the previous assignments for a detailed description of the scene file. In this section, only the elements introduced in this homework are explained.

- **Textures:**

  All of the specifications regarding texture mapping are defined under this element. It contains two types of child elements: Images and TextureMap. Images are simply a list of image files. An example is below:

```
<Images>
    <Image id="1">textures/galaxy.jpg</Image>
    <Image id="2">textures/rap01.jpg</Image>
</Images>
```

  Separating images from texture maps allows one to use the same image for different texture maps, avoiding memory duplication. For example, one mesh can use an image with nearest-neighbor sampling and another can use the same image with bilinear sampling.

  TextureMaps define the properties of actual texture mapping. An example is given below:

```
<TextureMap id="1" type="image">
    <ImageId>2</ImageId>
    <DecalMode>replace_kd</DecalMode>
    <Interpolation>bilinear</Interpolation>
</TextureMap>
```

  This defines that the texture map with id 1 is an image texture according to its type attribute. The other two possible types are *perlin* and *checkerboard*. You can of course extend this list, but these are the values we use for the homework.

  - **ImageId:** an index into the images element. Only texture maps whose type is *image* use this element.

- **DecalMode:** the mode of texture mapping. The possible values are the following:
    - replace_kd: Use the texture value as k_d value. Currently, divide the texture color value by 255 to obtain a three channel k_d value between $[0, 1]$ for each component. Textures defined like this are called diffuse textures.
    - blend_kd: Equally mix the existing material k_d value with the one that comes from the texture.
    - replace_ks: Use the texture value as k_s values. Currently, divide the texture color value by 255 to obtain a three channel k_s value between $[0, 1]$ for each component. Textures defined like this are called specular textures.
    - replace_background: This texture defines the background texture of the scene. You can think of this texture as the texture of the near plane. Only one texture can be defined as the background texture. If this texture is defined, the BackgroundColor element is ignored.
    - replace_normal: Use this texture for normal mapping. Note that this texture is defined in the canonical tangent space and the normal vector that it defines must be transformed to the actual tangent space of the surface.
    - bump_normal: Use this texture for bump mapping.
    - replace_all: Replace all components (i.e. diffuse, specular, and ambient) of the surface shading color with this texture's value.
- **Interpolation:** Defines whether nearest or bilinear interpolation should be used. If bilinear interpolation is defined and you are fetching a texel value from an edge pixel, you can repeat this edge pixel to avoid out-of-bounds access.
- **BumpFactor:** In bump mapping, it may be desirable to control the strength of the displacement function. This value determines this multiplication factor. In other words, you must multiply your displacement function with the value of this element to compute the real displacement.
- **NoiseScale:** This element only applies to Perlin noise textures. It is used to control the frequency of the Perlin noise. For implementation you must multiply the input position that you give to the noise function with this value. Smaller scales will create a low frequency noise texture and larger scales will create a high frequency noise texture.
- **NoiseScale:** This element only applies to Perlin noise textures. It defines whether you should take the absolute value (absval) of the noise function or whether you should linearly convert it (linear) to $[0, 1]$ range. If you take the absolute value, you get the snake-like patterns. Otherwise, you get a more patch noisy result.

  In addition to the image and Perlin textures, there is another procedural texture called the *checkerboard* texture that comes handy in certain scenarios. For example, the ground plane in the Veach-Ajar scene is defined using this texture. A checkerboard texture can be created using the following pseudo-code:

```
bool x = (int) ((pos.x + offset) * scale) % 2;
bool y = (int) ((pos.y + offset) * scale) % 2;
bool z = (int) ((pos.z + offset) * scale) % 2;

bool xorXY = x != y;
```

```
if (xorXY != z)
    return black;
else
    return white;
```

Here, the `pos` variable is the 3D point at which we are calculating the texture value. `offset` controls the starting point of the pattern. By changing the offset, you can shift the pattern in the 3D space. `scale` controls the size of the checkerboard squares. `black` and `white` define the colors of the checkerboard squares – they do not have to be black and white: you can define a purple and orange checkerboard if it suits your taste. These parameters are defined using the following XML elements:

- **Scale:** the `scale` variable defined above
- **Offset:** the `offset` variable defined above
- **BlackColor:** a color triplet defines the color of the "black" squares
- **WhiteColor:** a color triplet defines the color of the "white" squares

Up to know, we discussed the definition of textures. How can we actually use them for texture mapping on meshes? All object types we learned so far (Triangle, Sphere, Mesh, and MeshInstance) now also accept a new element called *Textures*. This element simply defines which TextureMaps defined previously in the file should be used for texture mapping on that object. Note that the Textures element can contain multiple values. For example, if we want to define a diffuse map, specular map, and a bump map for an object its Textures element may look like this:

```
<Textures>diffuse-map-id specular-map-id normal-map-id</Textures>
```

Of course, it is up to the designer to avoid using contradictory values in this element. For example, we shouldn't define a normal map and a bump map for an object. The inputs given to you should be free of such ambiguous definitions.

4 Hints & Tips

In addition to those for the previous homeworks, the following tips may be useful for this homework.

1. Although this homework is not difficult from an algorithmic point of view, it can take more time compared to the previous homework. So start early and ask questions if you run into issues.

5 Bonus

I will be more than happy to give bonus points to students who make important contributions such as new scenes, importers/exporters between our XML format and other standard file formats. Note that a Blender exporter[1], which exports Blender data to our XML format, was written by one of our previous students. You can use this for designing a scene in Blender and exporting it to our file format.

[1] `https://saksagan.ceng.metu.edu.tr/courses/ceng477/student/ceng477exporter.py`

6 Regulations

1. **Programming Language:** C/C++ is the recommended language. However, other languages can be used if so desired. In the past, some some students used Rust or even Haskell for implementing their ray tracers.

2. **Changing the Sample Codes:** You are free to modify any sample code provided with this homework.

3. **Additional Libraries:** If you are planning to use any library other than *(i)* the standard library of the language, *(ii)* pthread, *(iii)* the XML parser, and the PNG libraries please first ask about it on ODTUClass and get a confirmation. Common sense rules apply: if a library implements a ray tracing concept that you should be implementing yourself, do not use it!

4. **Submission:** Submission will be done via ODTUClass. To submit, Create a "**tar.gz**" file named "raytracer.tar.gz" that contains all your source code files and a Makefile. The executable should be named as "raytracer" and should be able to be run using the following commands (scene.xml will be provided by us during grading):

   ```
   tar -xf raytracer.tar.gz
   make
   ./raytracer scene.xml
   ```

   Any error in these steps will cause point penalty during grading.

5. **Late Submission:** You can submit your codes up to 3 days late. Each late day will cause a 10 point penalty.

6. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations and will get 0 from the homework. You can discuss algorithmic choices, but sharing code between groups or using third party code is strictly forbidden. By the nature of this class, many past students make their ray tracers publicly available. You must refrain from using them at all costs.

7. **Forum:** Check the ODTUClass forum regularly for updates/discussions.

8. **Evaluation:** The basis of evaluation is your blog posts. Please try to create interesting and informative blog posts about your ray tracing adventures. You can check out various past blogs for inspiration. However, also expect your codes to be compiled and tested on some examples for verification purposes. So the images that you share in your blog post must directly correspond to your ray tracer outputs.