RSO Report #2 – Environment Map Sampling

Jakub Profota

November 18, 2025

Source Code

Loading Environment Map HDR

First, a process of selecting environment map sampling was integrated into the existing source code, which is done by pressing the E key. There are many added switch case statements and minor code changes, which are not listed in this report for the sake of brevity. Mainly, however, a new structure representing the environment map light source was introduced and added to the scene.

```
// The light source represented by an environment map
struct InfiniteAreaLight {
  struct EnvMap {
    std::vector<vec3> image;
    int width, height;

    vec3 Lookup(const int w, const int h) const {
      return image[h * width + w];
    }

    float Y(const vec3 rgb) const {
      return 0.2126f * rgb.x + 0.7152f * rgb.y + 0.0722 * rgb.z;
    }
  } envMap;
  // ...
};
// ...
// The scene definition with main rendering method
class Scene {
  std::vector<Intersectable *> objects;
  double totalPower;
  int nLightSamples, nBRDFSamples, nInfiniteSamples;

public:
  Camera camera;
  InfiniteAreaLight infiniteAreaLight;
  // ...
};
```

In the build method of the scene, an environment map is loaded and preprocessed to be used in the sampling.

```
void Scene::build() {
  // ...
  // Load environment map
  if (!loadHDR(envMapFile, infiniteAreaLight.envMap.image,
               infiniteAreaLight.envMap.width,
               infiniteAreaLight.envMap.height))
    abort();
```

```
  infiniteAreaLight.ComputeRadianceMap();
  infiniteAreaLight.NormalizeRadianceMap();
  infiniteAreaLight.ComputeDistributions();
}
```

The HDR load procedure has been constructed by reversing the existing HDR save logic. It has been tested by first loading the HDR, saving that and then checking if the result is the same as the original loaded HDR.

```
bool loadHDR(const char *filename, std::vector<vec3> &data, int &width,
             int &height) {
  FILE *fp = fopen(filename, "rb");
  if (!fp) return false;
  char line[256];

  if (!fgets(line, sizeof(line), fp) || strncmp(line, "#?", 2) != 0) {
    fclose(fp); return false;
  }

  while (fgets(line, sizeof(line), fp)) {
    if (line[0] == '\n') continue;
    if (line[0] == '-') break;
  }

  if (sscanf(line, "-Y %d +X %d", &height, &width) != 2) {
    fclose(fp); return false;
  }

  std::vector<RGBE> rgbeData(width * height);
  size_t count = fread(rgbeData.data(), sizeof(RGBE), rgbeData.size(), fp);
  fclose(fp);

  if (count != rgbeData.size()) return false;
  data.resize(width * height);
  for (int i = 0; i < width * height; ++i) {
    float r, g, b;
    rgbeToRGB(rgbeData[i], r, g, b);
    data[i].x = r;
    data[i].y = g;
    data[i].z = b;
  }
  return true;
}
```

Preprocessing Environment Map

First, a radiance map was obtained by computing a perceived brightness for each pixel from its RGB value.

```
struct InfiniteAreaLight {
  // ...
  std::vector<std::vector<float>> map;
  void ComputeRadianceMap() {
    map.resize(envMap.width);
    for (auto &column : map) column.resize(envMap.height);
    for (int w = 0; w < envMap.width; w++)
      for (int h = 0; h < envMap.height; h++)
        map[w][h] = envMap.Y(envMap.Lookup(w, h));
  }
  // ...
}
```

The radiance map was then normalized by first multiplying each pixel with a sine function, thus reducing the luminance of the pixel the closer it is to one of the poles, and then dividing each pixel by the total luminance of the environment map. Now, each pixel holds a value representing the probability (directly corresponding to the total luminance of the pixel) of choosing such pixel during the sampling method. The sum of values of all pixels is thus 1.

```
struct InfiniteAreaLight {
  // ...
  std::vector<std::vector<float>> normMap;
  void NormalizeRadianceMap() {
    float totalLuminance = 0.0f;
    normMap.resize(envMap.width);
    for (auto &column : normMap) column.resize(envMap.height);
    for (int h = 0; h < envMap.height; h++) {
      float sin = std::sin(M_PI * float(h + 0.5f) / float(envMap.height));
      for (int w = 0; w < envMap.width; w++) {
        normMap[w][h] = map[w][h] * sin;
        totalLuminance += normMap[w][h];
      }
    }
    for (int h = 0; h < envMap.height; h++)
      for (int w = 0; w < envMap.width; w++)
        normMap[w][h] /= totalLuminance;
  }
  // ...
}
```

Finally, to be able to efficiently sample the 2D image, a probability distribution is computed for each column of the image. To choose the corresponding column, a probability distribution of column distributions is also computed.

```
struct InfiniteAreaLight {
  // ...
  struct Distribution {
    std::vector<float> cdf;
    float funcInt, invFuncInt, invCount;
    int count;

    void ComputeCDF(std::vector<float> &f) {
      count = static_cast<int>(f.size());
      cdf.resize(count + 1);
      cdf[0] = 0.0f;
      for (int i = 0; i < count; ++i) cdf[i + 1] = cdf[i] + f[i];
      funcInt = cdf[count];
      invFuncInt = (funcInt > 0.0f) ? 1.0f / funcInt : 0.0f;
      invCount = 1.0f / static_cast<float>(count);
      for (int i = 1; i <= count; ++i) cdf[i] *= invFuncInt;
    }
    // ...
  };
  // ...
  void ComputeDistributions() {
    hDists.resize(envMap.width);
    for (int w = 0; w < envMap.width; w++)
      hDists[w].ComputeCDF(normMap[w]);
    std::vector<float> funcInts(envMap.width);
    for (int w = 0; w < envMap.width; w++)
      funcInts[w] = hDists[w].funcInt;
    wDist.ComputeCDF(funcInts);
  }
}
```

Sampling Environment Map

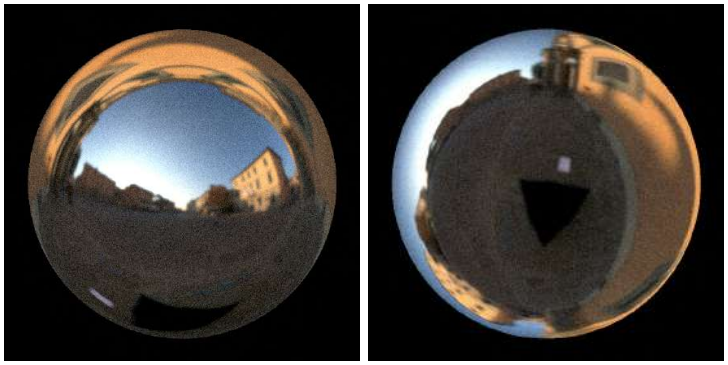
In the trace method of the scene, another for loop was added for the samples of the light source represented by the environment map. First, a light vector and the probability of choosing it is sampled from the preprocessed environment map, and both values are then fed into the BRDF of the material of the hit object to compute the radiance contribution.

```
  double riasX = 0.0, riasY = 0.0, riasZ = 0.0;
#pragma omp parallel for reduction(+ : riasX, riasY, riasZ)
  for (int i = 0; i < nInfiniteSamples; i++) {
    vec3 outDir;
    vec3 radiance_Li = infiniteAreaLight.Sample_L(outDir);
    double pdfLight = infiniteAreaLight.PDF(outDir);
    if (pdfLight < epsilon)
      continue;
    double cosThetaSurface = dot(hit.normal, outDir);
    if (cosThetaSurface > 0.0) {
      vec3 brdf = hit.material->BRDF(hit.normal, inDir, outDir);
      vec3 f = radiance_Li * brdf * cosThetaSurface;
      vec3 contribution = f / pdfLight / nTotalSamples;
      riasX += contribution.x;
      riasY += contribution.y;
      riasZ += contribution.z;
    }
  }
  radianceInfiniteAreaSampling = vec3(riasX, riasY, riasZ);
```

To speed up the computation, OpenMP was utilized to parallelize on multiple threads. However, the implementation of the random number function had to be changed to a thread-safe variant to make it work correctly.

```
// Compute random number in range [0,1], uniform distribution
double drandom() {
  thread_local std::mt19937 generator(
      std::random_device{}() +
      std::hash<std::thread::id>{}(std::this_thread::get_id()));
  thread_local std::uniform_real_distribution<double> distribution(0.0,
    1.0);
  return distribution(generator);
}
```

The light direction is sampled by generating two random numbers, which are used to choose the column and then the pixel of the image using the precomputed probability distributions. The sampled direction is rotated in a specific way. First, it ensures that the point in the center of the environment map image is roughly right behind the camera (and thus the reflection we see in the center of the sphere is roughly the center of the environment map image). Second, it ensures that the ground of the environment map is reflected on the lower part of the sphere (that also inherently flips the environment map, since the original HDR is upside down). This rotation made it easy to debug and check the correctness, see the rotated (left) and unrotated variants below.



```
struct InfiniteAreaLight {
  // ...
  vec3 Sample_L(vec3 &L) {
    // Find sample coordinates
    float fu = wDist.Sample(drandom());
    int u = static_cast<int>(fu);
    u = std::min(std::max(0, u), envMap.width - 1);
    float fv = hDists[u].Sample(drandom());
    int v = static_cast<int>(fv);
    v = std::min(std::max(0, v), envMap.height - 1);
    float theta = fv * hDists[u].invCount * M_PI;
    float phi = fu * wDist.invCount * 2.0f * M_PI;
    float cosTheta = cos(theta); float sinTheta = sin(theta);
    float cosPhi = cos(phi); float sinPhi = sin(phi);
    vec3 L_local = vec3(sinTheta * cosPhi, sinTheta * sinPhi, cosTheta);
    // Rotate the vector so we don't look to the ground
    L.x = -L_local.y;
    L.y = -L_local.z;
    L.z = -L_local.x;
    return envMap.image[v * envMap.width + u];
  }

  double PDF(const vec3 L) const {
    // Rotate the vector back
    vec3 L_local = vec3(-L.z, -L.x, -L.y);
    double theta = std::acos(std::max(-1.0, std::min(1.0, L_local.z)));
    double phi = std::atan2(L_local.y, L_local.x);
    if (phi < 0.0) phi += 2.0 * M_PI;
    double sinTheta = std::sin(theta);
    if (sinTheta == 0.0) return 0.0;
    int u = static_cast<int>(phi * (1.0 / (2.0 * M_PI)) * envMap.width);
    u = std::max(0, std::min(envMap.width - 1, u));
    int v = static_cast<int>(theta * (1.0 / M_PI) * envMap.height);
    v = std::max(0, std::min(envMap.height - 1, v));
    float prob_uv = normMap[u][v];
    double pdf = (prob_uv * envMap.width * envMap.height) /
                 (2.0 * M_PI * M_PI * sinTheta);
    return pdf;
  }
}
```

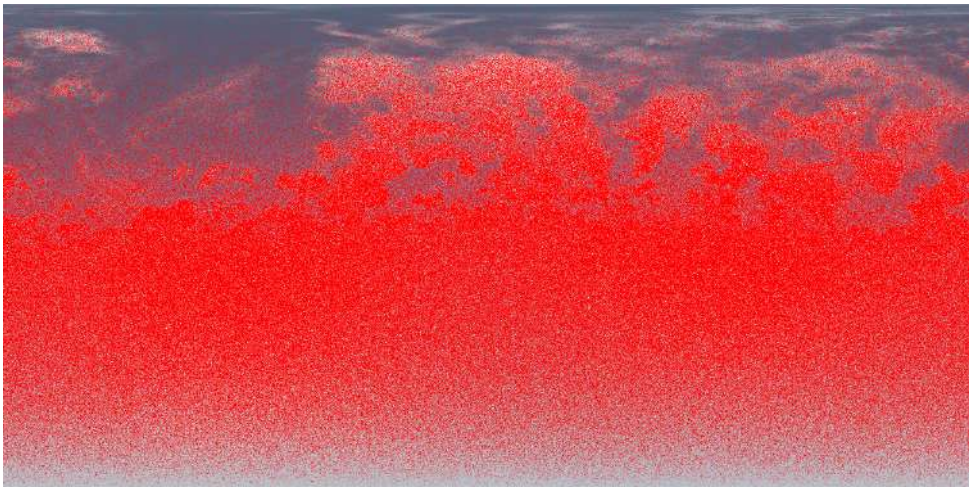
The final piece of the puzzle is the probability distribution sampling method.

```
struct InfiniteAreaLight {
  // ...
  struct Distribution {
    // ...
    float Sample(float u) { // Parameter in [0,1] range
      auto it = std::lower_bound(cdf.begin(), cdf.end(), u);
      std::ptrdiff_t d = std::distance(cdf.begin(), it) - 1;
      if (d < 0) d = 0;
      if (d > count - 1) d = count - 1;
      int offset = static_cast<int>(d);
      float cdf0 = cdf[offset];
      float cdf1 = cdf[offset + 1];
      float du = (cdf1 > cdf0) ? (u - cdf0) / (cdf1 - cdf0) : 0.0f;
      return (offset + du);
    }
  };
  // ...
}
```

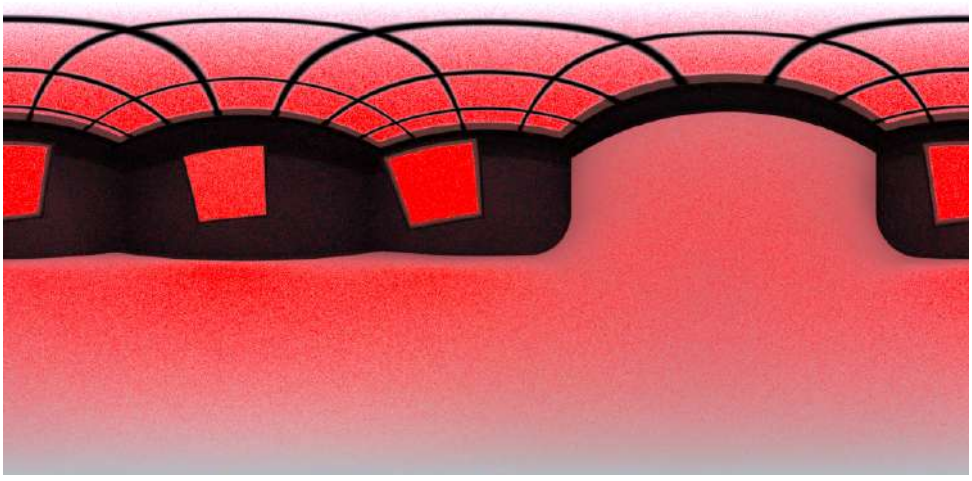
Results

Sampling Visualizations

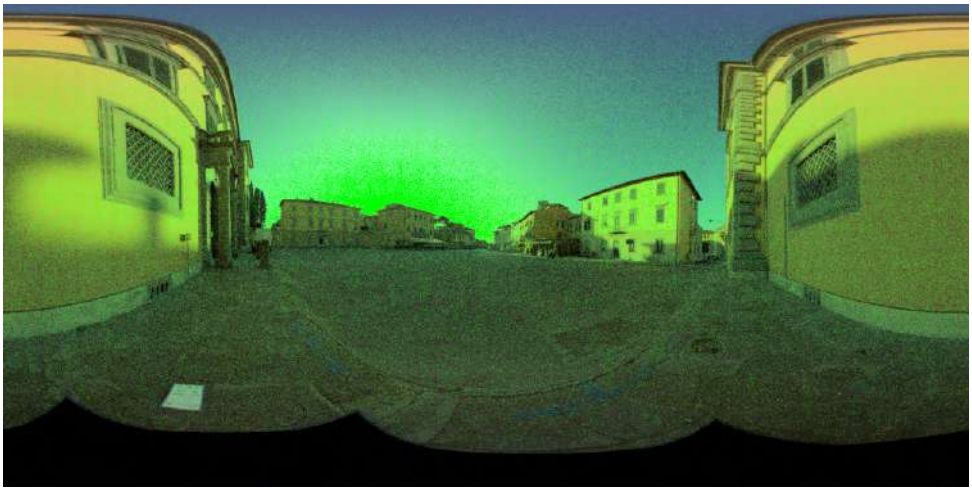
The following images visualize the picked samples with a bright red or bright green color. It is clearly visible that the dense clusters of red or green color end up in the regions with comparatively high luminance. Also, pixels close to the poles of the environment map (EM) are picked less often (see for example EM 004). These visualizations demonstrate the correctness of normalization and sampling of the environment maps. All visualization images were converted from the HDR format to PNG with default tonemapping options of the *tev* HDR viewer, that is exposure +0.0, offset +0.0 and gamma +2.2.



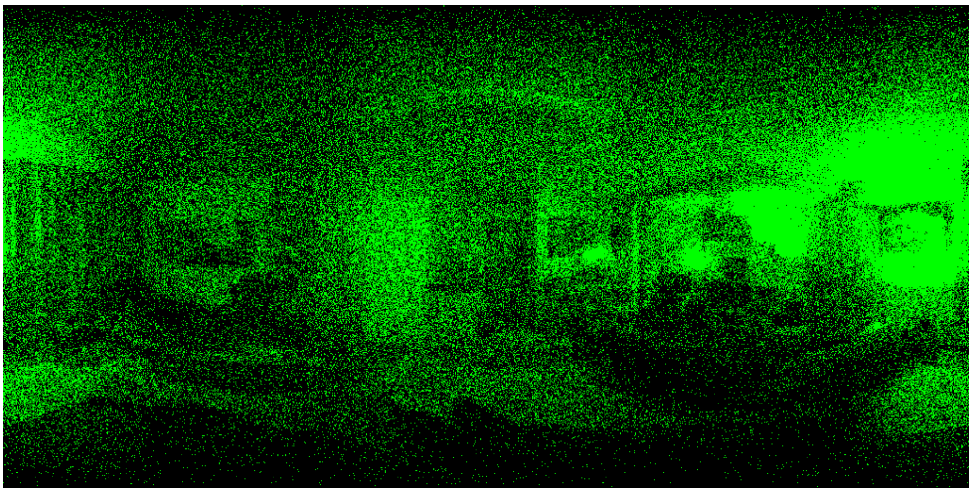
EM 001 with resolution of 2100×1050 pixels and 10 samples per pixel.



EM 004 with resolution of 6000×3000 pixels and 100 samples per pixel.



EM 007 with resolution of 4096×2048 pixels and 10 samples per pixel.

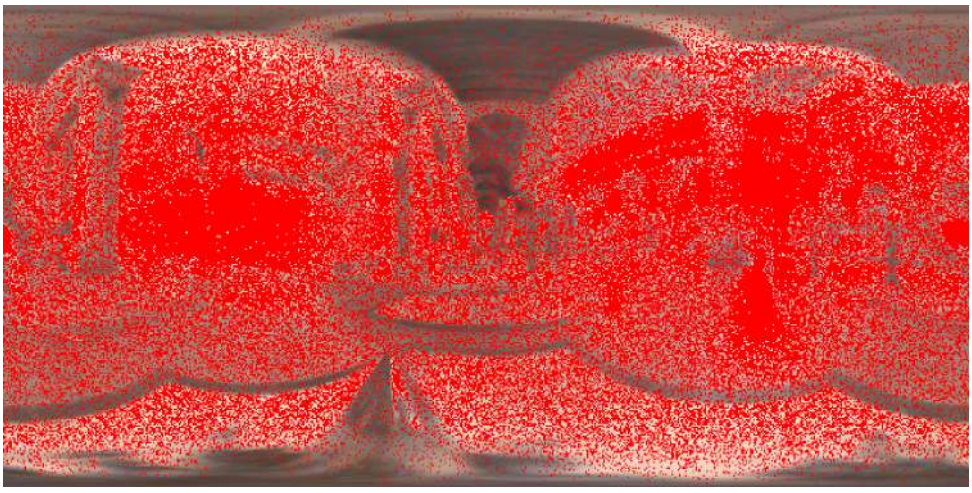


EM 010 with resolution of 1024×512 pixels and 2 samples per pixel.

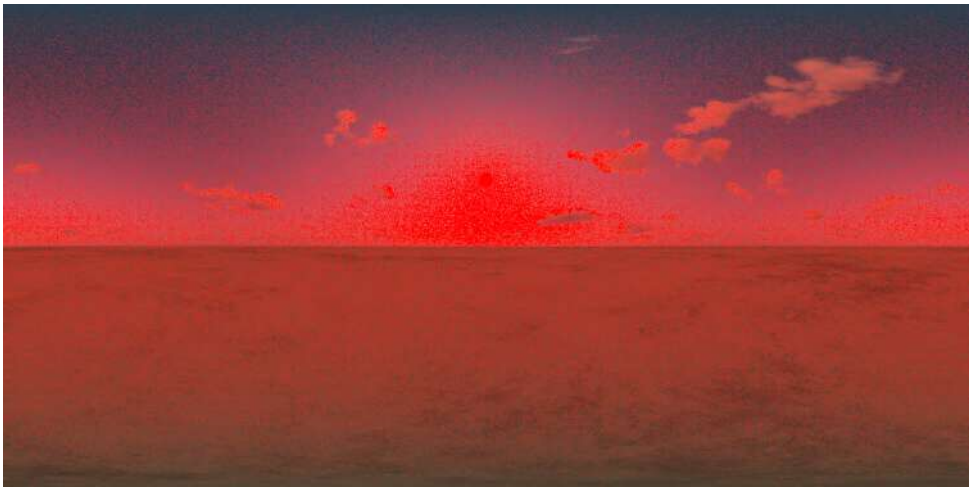
EM 011 with resolution of 2400×1200 pixels and 10 samples per pixel.

EM 013 with resolution of 2048×1024 pixels and 10 samples per pixel.

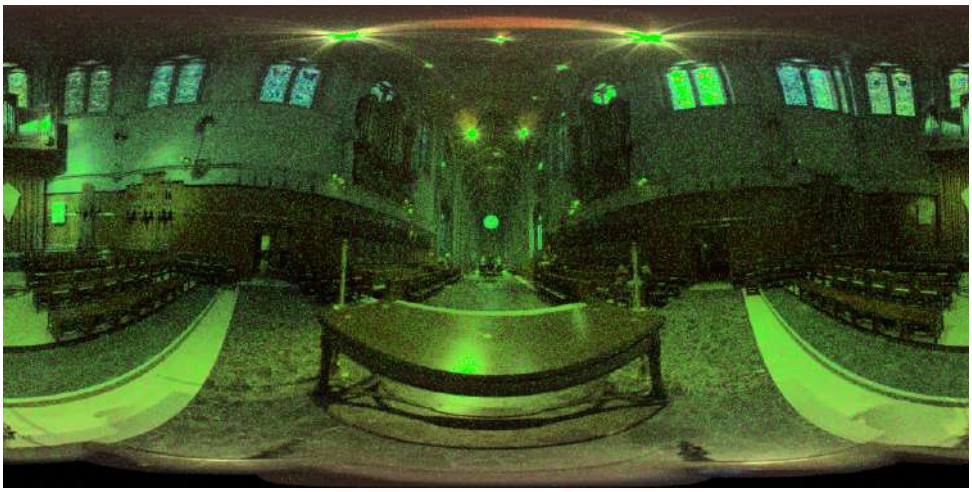
EM 015 with resolution of 2048×1024 pixels and 2 samples per pixel.



EM 016 with resolution of 639×319 pixels and 2 samples per pixel.



EM 020 with resolution of 4096×2048 pixels and 50 samples per pixel.



EM 023 with resolution of 3072×1536 pixels and 10 samples per pixel.

EM 024 with resolution of 2048×1024 pixels and 20 samples per pixel.

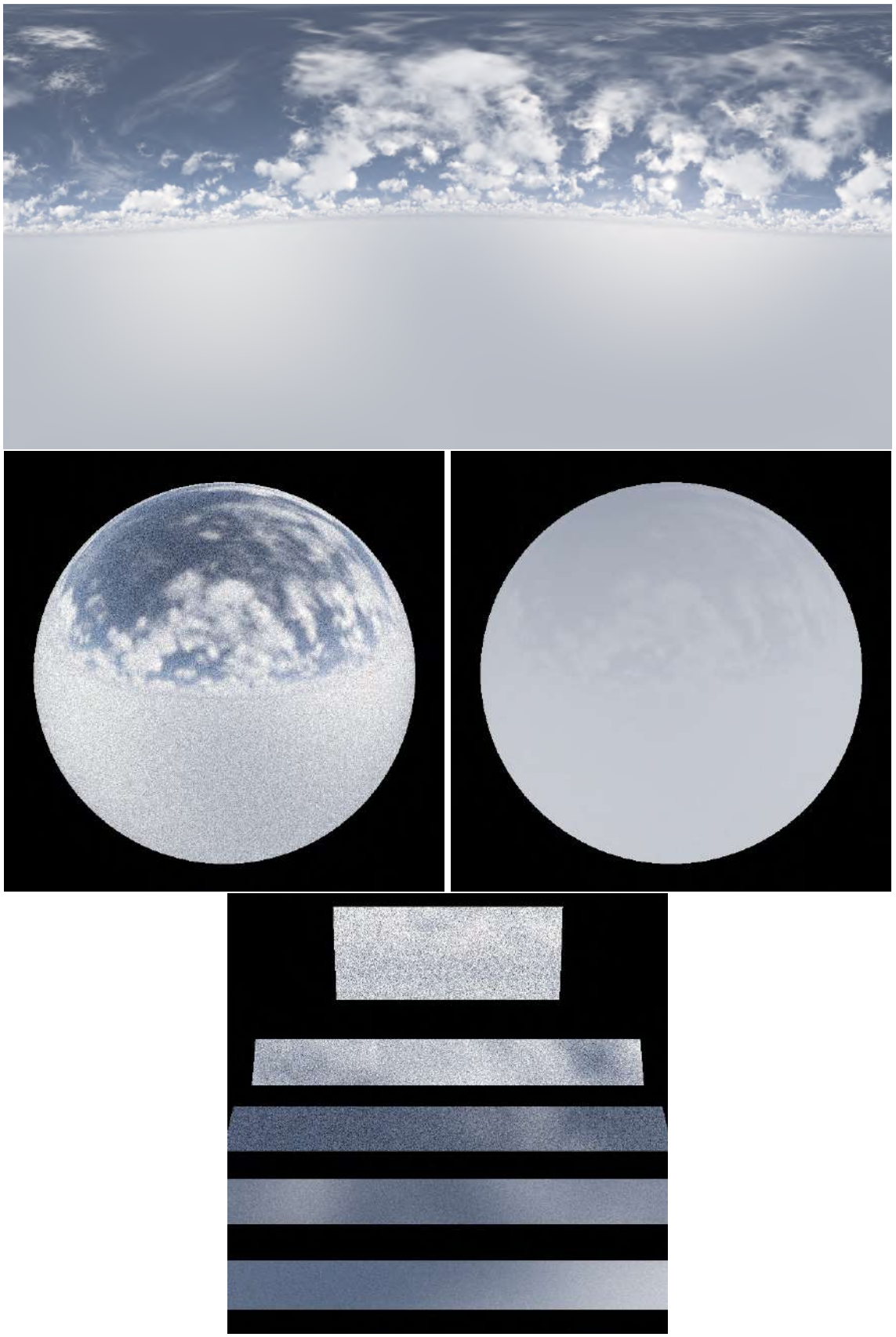
Computed Images

For each environment map, three images were computed. First of a specular sphere with specular albedo $= (1, 1, 1)$ and shininess factor of $5,000$. Second of a mostly diffuse sphere with diffuse albedo $= (0.9, 0.9, 0.9)$, specular albedo $= (0.1, 0.1, 0.1)$ and shininess factor of $5,000$. And the third image is of the four original rectangles plus a new one oriented in such a way that it roughly mirrors the center of the environment map with shininess factor of $20,000$. To get a perfect mirror, one would set a shininess factor close to infinity to only reflect perpendicular light samples, but it would require much much higher computation effort to obtain a representative image. Even now it is clearly visible that the top rectangle with shininess factor of $20,000$ is much sharper but also much more noisier than the bottom-most rectangle with shininess factor of just 500. The sphere and the rectangle were constructed with the following code snippet.

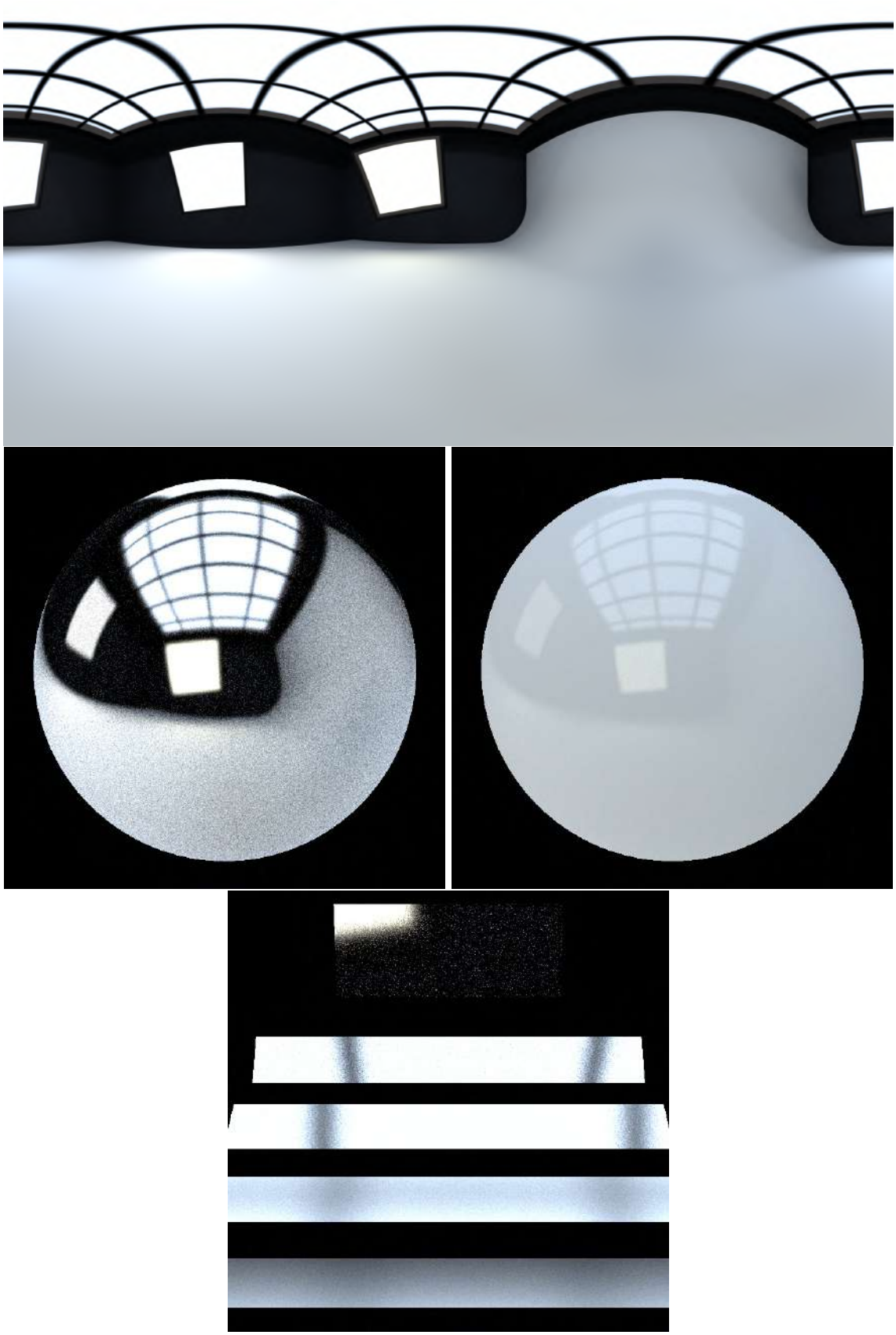
```
objects.push_back(new Sphere(vec3(0, 0, 0), 5, new TableMaterial(5000)));
objects.push_back(new Rect(vec3(0, 0, -45), eyePos, lightCenterPos, 10, 4,
                           new TableMaterial(20000)));
```

Each of the following images were computed in 10 iterations with $10,000$ samples per each iteration for a grand total of $100,000$ samples. Unless explicitly stated otherwise, the default tonemapping options of the *tev* HDR viewer, that is exposure $+0.0$, offset $+0.0$ and gamma $+2.2$, were used to convert the HDR image to PNG.

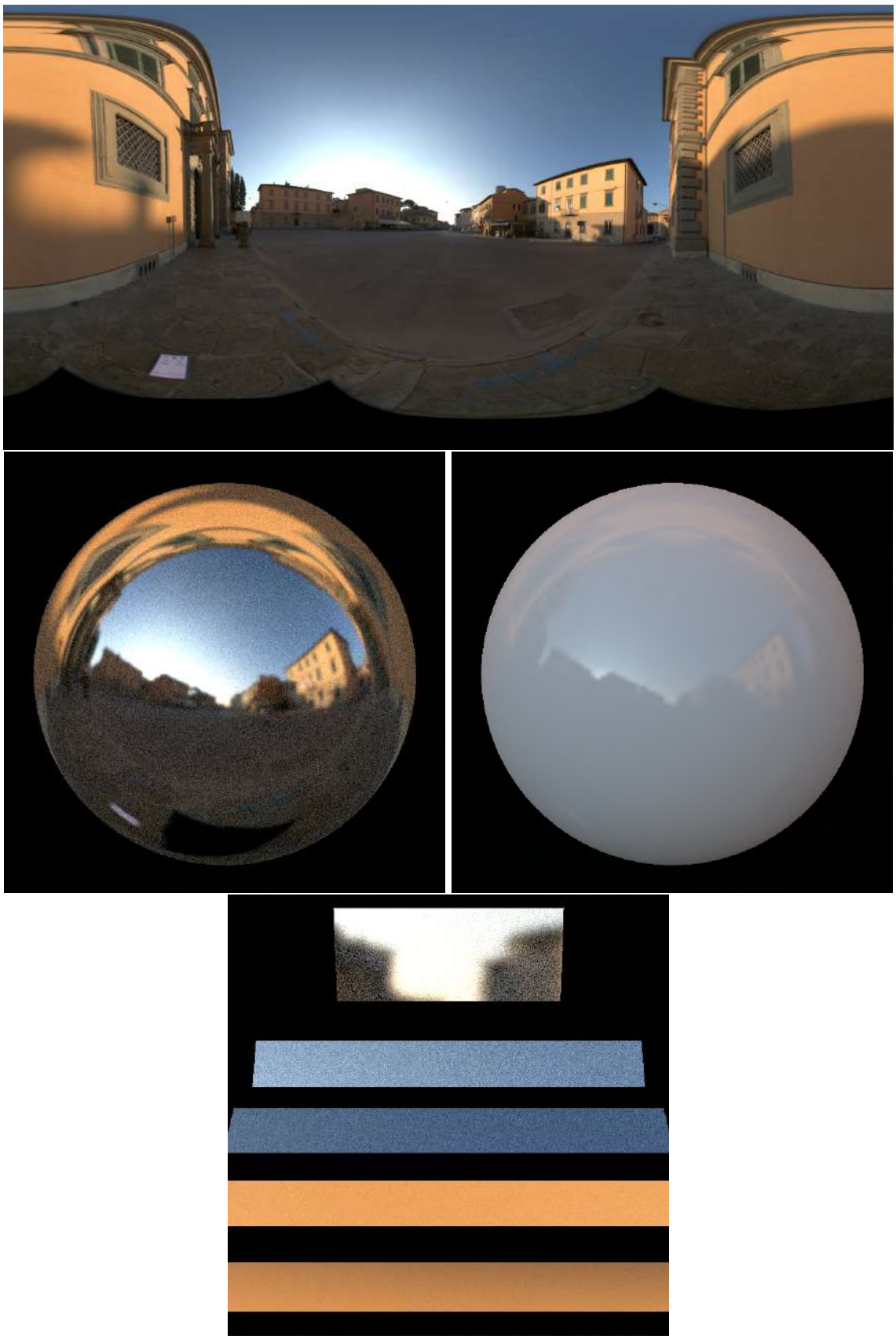
EM 001.



EM 004.

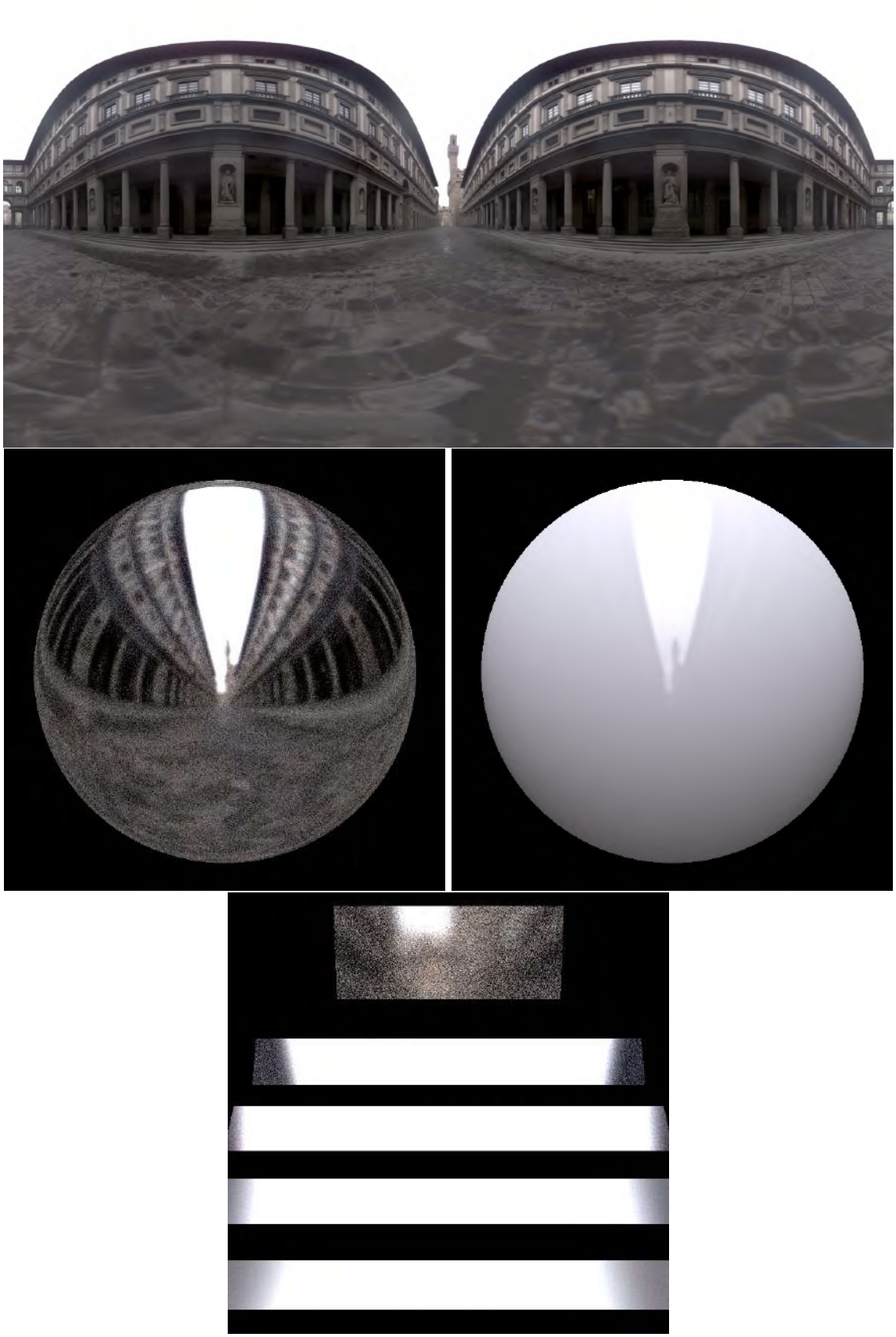


EM 007.

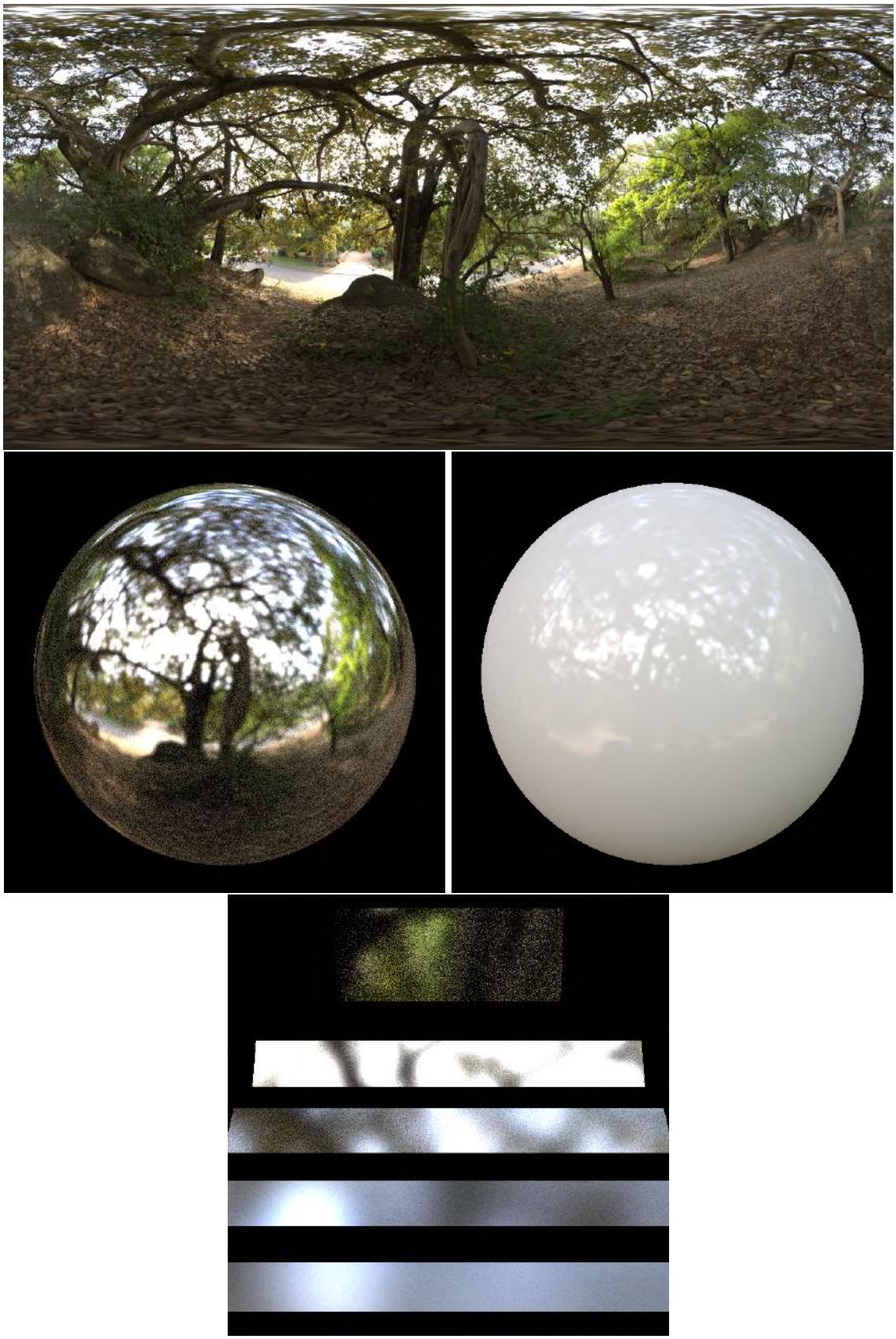


EM 010. Exposure +5.0, offset +0.0, gamma +3.5.

EM 011. Exposure −1.2, offset +0.0, gamma +2.2.

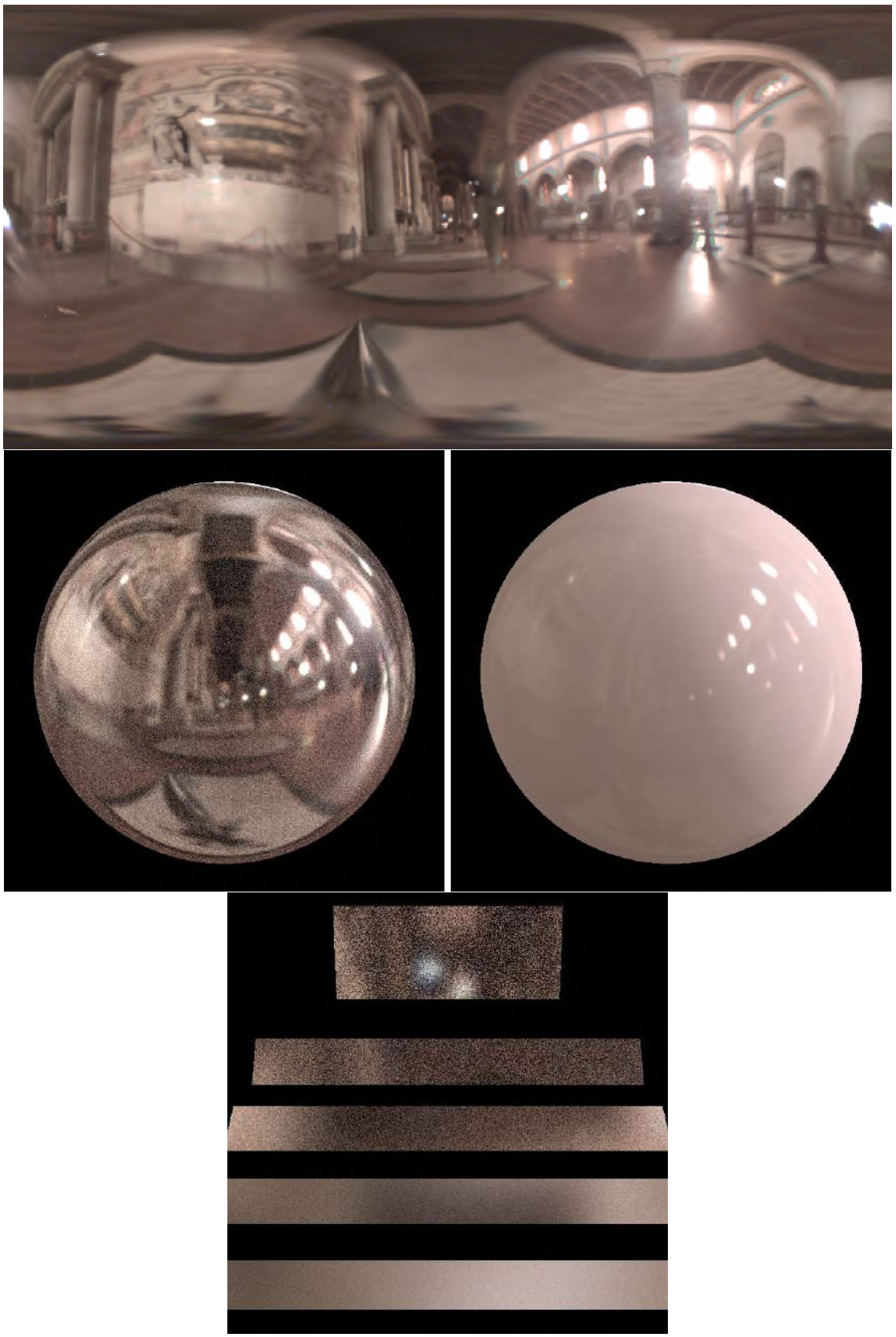


EM 013. Exposure −0.5, offset +0.0, gamma +2.2.

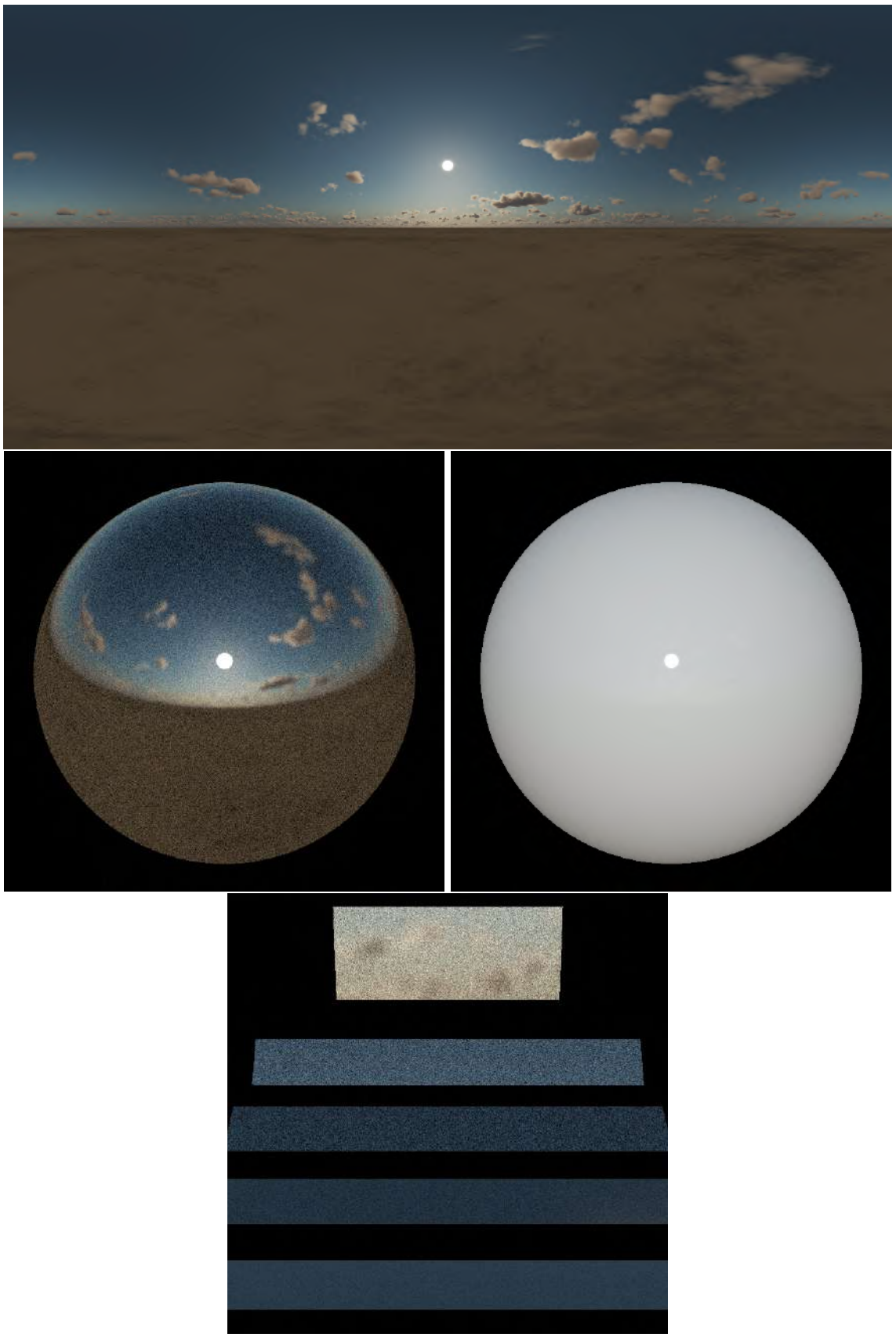


EM 015. Exposure +2.0, offset +0.0, gamma +2.2.

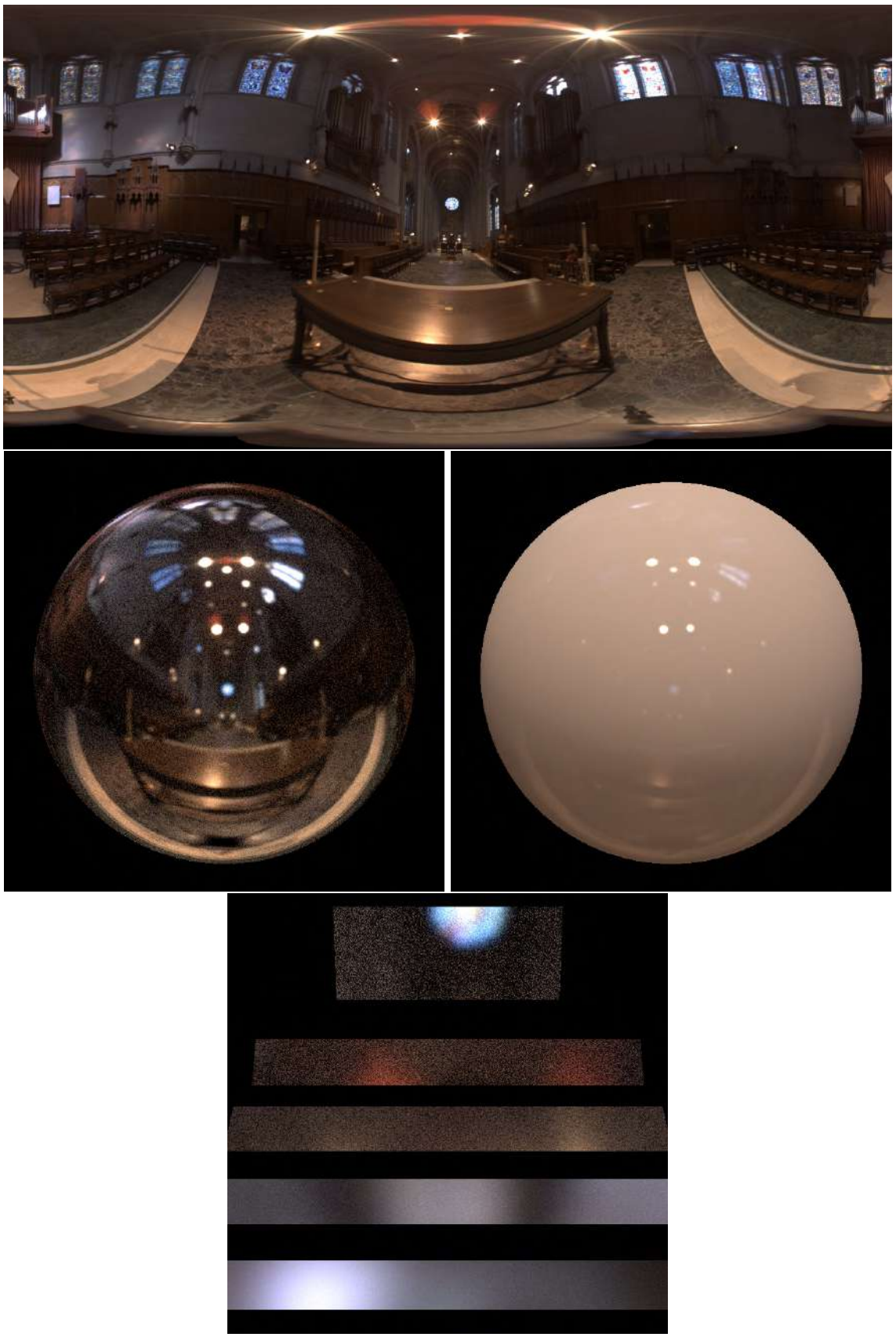
EM 016. Exposure −1.5, offset +0.0, gamma +2.2.



EM 020. Exposure −0.5, offset +0.0, gamma +2.2.



EM 023.



EM 024.