Real-Time Triangle Shading and Rendering with Gouraud and Phong Lighting in Python

Dimitrios Karatis 10775, *Electrical and Computer Engineering, AUTH*

***Abstract*—This project extends a 3D graphics rendering pipeline by implementing realistic lighting and shading techniques. It introduces the Phong reflection model through a custom material class and computes lighting using ambient, diffuse, and specular components. The system supports both Gouraud and Phong shading for triangle rasterization, using texture sampling and barycentric interpolation. The implementation includes accurate handling of vertex normals, UV coordinates, and per-vertex lighting computations. The results demonstrate visually correct rendering under different lighting setups, enabling realistic visualization of 3D surfaces. All functionality is implemented in Python using NumPy.**

***Index Terms*—computer graphics, Phong lighting, Gouraud shading, shading models, 3D rendering, triangle rasterization, lighting models**

A. `MatPhong` class

The `MatPhong` class encapsulates the material properties required for Phong lighting calculations. It is initialized with four parameters: the ambient reflection coefficient `ka`, the diffuse reflection coefficient `kd`, the specular reflection coefficient `ks`, and the Phong exponent `n`. These parameters control how the surface of an object interacts with light. Specifically, `ka` determines how much ambient light the material reflects uniformly in all directions. `Kd` how strongly the surface reflects diffuse light depending on the angle between the light source and surface normal. `Ks` controls the intensity of specular highlights, and `n` affects the sharpness of the specular reflection, with higher values producing shinier surfaces.

Detailed code for this function can be found in the MatPhong.py file.

B. `light` function

The `light` function implements the Phong illumination model, which computes the final RGB color of a surface point based on material properties, lighting conditions, and viewer position. It receives as input the surface point and its normal vector, the texture color at the point, the camera position, the ambient light, and one or more point light sources with their respective positions and intensities. It also supports optional precomputed view and light directions (used in Phong shading to avoid recalculating per pixel). The final color is computed as the sum of ambient, diffuse, and specular contributions from all light sources. Ambient light is modulated by the material's ambient coefficient. Diffuse light depends on the angle between the surface normal and light direction, and specular highlights depend on the reflection direction and view vector, raised to a Phong exponent that controls shininess.

This document was prepared by Dimitrios Karatis as part of the third assignment for the Computer Graphics course at Aristotle University of Thessaloniki.

Testing and Evaluation

To test the function, three separate scenarios were considered. First, a single white light source was placed directly above a gray surface point facing upward, with a viewer also positioned above. The result was a white color vector `[1.0, 1.0, 1.0]`, indicating full brightness due to perfect alignment of light, normal, and view vectors. Next, three light sources of different intensities were placed above the point in various directions. The resulting color, `[0.8158, 0.8158, 0.8158]`, confirmed correct aggregation of lighting contributions. Finally, the same setup was used with fixed view and light directions, as typically done in Phong shading. The resulting color, `[0.5844, 0.5844, 0.5844]`, showed reduced intensity due to angular deviation from the normal direction.

Function conclusion

The output of the tests is consistent with the expected behavior of the Phong model. Specular highlights increase when the view and light directions align with the surface normal, and the color remains within the valid RGB range $[0, 1]$.

Detailed code for this function can be found in the light_func.py file.

C. `calc_normals` function

The `calc_normals` function is responsible for computing smooth vertex normals for a triangle mesh. It receives two inputs: a $3 \times N_v$ array of 3D vertex coordinates and a $3 \times N_t$ array of triangle indices referencing the vertices. Each triangle is defined by three indices, and the function assumes 0-based indexing. For each triangle, the function computes a face normal using the cross product of two edge vectors. This face normal is then added to the accumulated normal vectors of each of the triangle's vertices. After all triangles are processed, the resulting vertex normals are normalized to unit length.

Testing and Evaluation

To verify the correctness of the implementation, the function was tested on a simple 3D object: a square-based pyramid with five vertices and eight triangles (four for the base and four for the sides). A visualization script using `matplotlib` was created to render the pyramid mesh and overlay the calculated vertex normals. In the plot, each vertex is shown in black, and the corresponding normal vector is drawn in red.

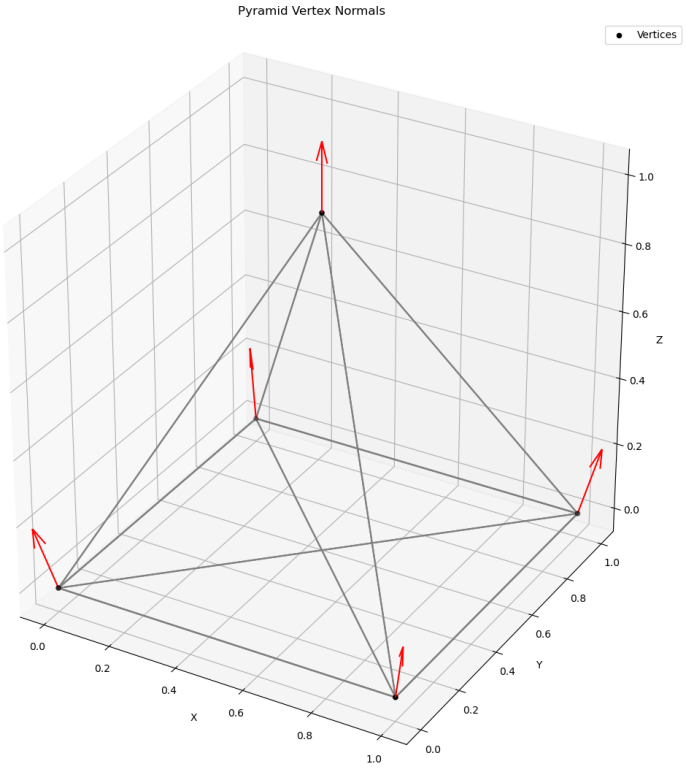

Fig. 1: calc_normals test

The output shows that the normals at the base corners point diagonally outward and upward, consistent with the average of the normals from the base and side faces. The normal at the apex of the pyramid points vertically upward. All normals have unit length and point in plausible directions that reflect the surrounding geometry.

Function conclusion

The test confirms that the function works as intended. The calculated normals appear correct, with smooth transitions between faces, making them suitable for use in shading models such as Gouraud or Phong shading.

Detailed code for this function can be found in the calc_normals_func.py file.

D. `shade_gouraud` function

The `shade_gouraud` function implements Gouraud shading for a single triangle using the Phong reflection model. For each of the triangle's three vertices, it calculates the lighting based on the vertex normal, view direction, and light sources, incorporating ambient, diffuse, and specular components according to the `MatPhong` material. The function interpolates the resulting RGB colors linearly across the triangle using barycentric coordinates. Texture sampling is performed per vertex using UV coordinates, and the final color is computed by combining the texture value and lighting result before rasterizing into the image buffer.

Testing and Evaluation

To evaluate the correctness of the shading, a test was performed using a triangle whose vertices have significantly different surface normals. All three vertices sample from the same red texture, but the lighting varies due to the orientation of each normal relative to the light source. One normal faces directly toward the light, while the others are tilted away. This setup creates a strong lighting contrast across the triangle and is designed to highlight the interpolation of vertex colors.

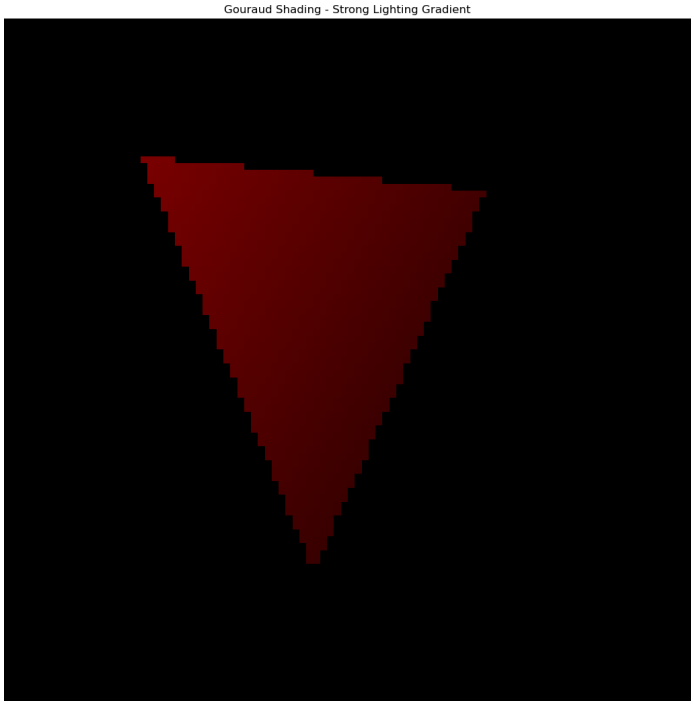

Fig. 2: shade_gouraud test

Function conclusion

The output exhibits a clear gradient in brightness across the triangle surface, with the vertex facing the light appearing brighter and the others darker. This matches the expected behavior of Gouraud shading, where lighting is computed per vertex and interpolated across the surface. The result confirms that the function correctly computes and blends lighting contributions.

Detailed code for this function can be found in the shade_gouraud_func.py file.

E. `shade_phong` function

The `shade_phong` function implements per-pixel Phong shading for a single triangle projected onto the image plane. It begins by computing the bounding box of the triangle and interpolates the normal vectors and texture coordinates for each pixel inside the triangle using barycentric coordinates. A fixed view vector $\vec{V}$ and light direction vectors $\vec{L}$ are computed once per triangle using the centroid of the triangle in 3D space, prior to projection, as required by the assignment specification. For each pixel, the interpolated normal vector is normalized and combined with the sampled texture color to compute the final illumination using the Phong lighting model, via the provided `light()` function. The pixel color is then written to the output image buffer.

Testing and Evaluation

To test the implementation, a single triangle was rendered using a synthetic setup: one red-colored texture, a directional camera, and a single light source. The camera and light source were placed in positions that allow for a clear view of diffuse and specular components. The material parameters of the triangle were set using an instance of the `MatPhong` class.

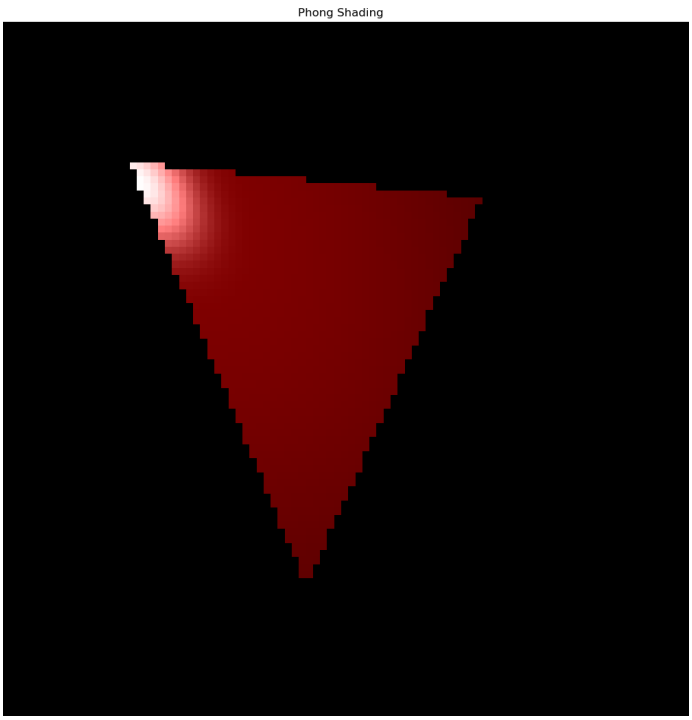

Fig. 3: shade_phong test

Function conclusion

The produced image shows a smooth red triangle with a bright specular highlight on its upper-left side. This indicates that the interpolation of normals and texture coordinates is working correctly and that the lighting is computed per pixel. The result matches the theoretical behavior of the Phong illumination model, as the brightness increases smoothly towards the light reflection direction. Therefore, the output verifies that the function behaves as intended.

Detailed code for this function can be found in the shade_phong_func.py file.

F. `render_object` function

The `render_object` function constitutes the core rendering pipeline for projecting a 3D textured object onto a 2D image plane. It begins by computing vertex normals using mesh connectivity, which are essential for lighting calculations. The camera transformation is performed using the `lookat` function, aligning the world with the camera coordinate system. The 3D vertices are then projected into 2D using a perspective projection model, followed by rasterization into pixel space, all by using functions from the previous assignment. For each triangle in the mesh, the appropriate shading method (*Gouraud* or *Phong*) is selected based on the input parameter, and then the triangle is shaded using interpolated normals and UV coordinates. The result is a rendered image where each triangle has been shaded according to the specified lighting model and material properties, producing photorealistic results.

Detailed code for this function can be found in the render_object_func.py file.

G. `demo` function

The `demo` script serves as the main driver for generating rendered images of a 3D object using the data provided in the `hw3.npy` file. It loads the geometric information, camera setup, lighting configuration, and material properties, and then applies both Phong and Gouraud shading techniques under different lighting scenarios: ambient-only, diffuse-only, specular-only, and full lighting. For each combination, it constructs a `MatPhong` material with the appropriate reflection coefficients and calls the `render_object` function to produce the image. The resulting images are saved with descriptive filenames that indicate the shader and lighting mode used.

Detailed code for this function can be found in the demo.py file.

H. `test_load` function

The `test_load` function serves as a utility for inspecting the contents of the `hw2.npy` file, which stores all necessary data for the 3D rendering task. It uses NumPy's `load` function with `allow_pickle=True` to deserialize the file into a Python dictionary. It then prints all the available keys in the dataset along with the data type and shape of each corresponding value. This diagnostic tool is especially useful for understanding the structure of the input data, verifying that it has been loaded correctly, and checking that all required parameters—such as vertex positions, triangle indices, camera intrinsics, and animation parameters—are present and properly formatted before being used in the rendering pipeline.

The `hw3.npy` file contains a Python dictionary with 18 key-value pairs that encapsulate all the necessary data for rendering a 3D scene. Camera configuration is defined by `cam_pos`, `up`, and `target` (all 3×1 `ndarray`), the physical dimensions of the image plane are given by `plane_h` and `plane_w` (both integers), while `res_h` and `res_w` define the pixel resolution of the output image. The focal length of the camera is provided via `focal` (int). The 3D object geometry is described by `v_pos` (3×1984 `ndarray` of vertex positions), `v_uvs` (1984×2 `ndarray` of texture coordinates), and `t_pos_idx` (960×3 triangle index array of type `trimesh.caching.TrackedArray`). Lighting parameters include `l_pos` and `l_int`, each a list of three 3D vectors representing the positions and RGB intensities of three point light sources, and `l_amb` (3D `ndarray`) representing ambient light intensity. The material properties required for Phong shading are given by scalar values `ka` (ambient coefficient), `kd` (diffuse coefficient), `ks` (specular coefficient), and `n` (Phong exponent).

Detailed code for this function can be found in the test_load_func.py file.

FUNCTIONS FROM THE PREVIOUS ASSIGNMENT

A. `world2view` function

The `world2view` function transforms 3D points from world coordinates into camera view coordinates, simulating the perspective of a virtual camera in a 3D scene. It takes three inputs: `pts`, a $3 \times N$ matrix (or alternatively an $N \times 3$ matrix) representing N 3D points in world coordinates; `R`, a 3×3 rotation matrix that defines the orientation of the camera; and `c0`, a 3D vector specifying the position of the camera in world coordinates. Internally, the function reshapes and verifies the input data as necessary and then applies the transformation formula:

$$\texttt{view\_coords} = R \cdot (\texttt{world\_coords} - \texttt{c0}),$$

which translates the points relative to the camera's position and then rotates them into the camera's reference frame. The output is returned as an $N \times 3$ matrix of points in camera coordinates.

Detailed code for this function can be found in the world2view_func.py file.

B. `lookat` function

The `lookat` function computes the camera's view transformation given the camera's position (`eye`), the up direction vector (`up`), and the target point (`target`) the camera is looking at. It returns a rotation matrix R and a translation vector t that together transform points from world coordinates to camera coordinates using the formula:

$$\text{camera\_coords} = R \times (\text{world\_coords} - t).$$

The function first calculates the camera's forward axis as the normalized vector pointing from the camera position to the target. Then, it computes the right axis as the normalized cross product between the forward vector and the up vector, ensuring orthogonality. The true up vector is recalculated as the cross product of the right and forward vectors to maintain an orthonormal basis. These three vectors form the rows of the rotation matrix R, which aligns the world coordinate system with the camera's coordinate system. The translation vector t represents the camera's position in world coordinates.

Detailed code for this function can be found in the lookat_func.py file.

C. `perspective_project` function

The `perspective_project` function performs the projection of 3D points onto a 2D image plane using the pinhole camera model. It takes as inputs a set of 3D points in world coordinates (`pts`), the camera's focal length (`focal`), a rotation matrix R that transforms points from world to camera coordinates, and a translation vector t representing the camera's position. The function first converts the 3D points to the camera coordinate system by applying the rotation and translation using the `world2view` function. It then extracts the depth values (distance along the camera's viewing axis) for each point. Finally, it computes the 2D projected coordinates on the image plane by scaling the x and y camera coordinates with the focal length and normalizing by the depth, thereby implementing the perspective projection. The function returns the 2D image coordinates as a $2 \times N$ matrix along with the corresponding depth values as a $1 \times N$ array.

Detailed code for this function can be found in the perspective_project_func.py file.

D. `rasterize` function

The `rasterize` function converts 2D points given in camera plane coordinates into corresponding pixel coordinates on a discrete image grid. It takes as input the 2D points in a continuous coordinate system centered at the camera plane, the physical width and height of the camera plane in world units, and the desired resolution of the output image in pixels (width and height). The function computes scale factors to map world units to pixels and shifts the points so that the origin of the camera plane aligns with the center of the image. Since the camera plane uses an upward positive y-axis while image pixel coordinates typically have the origin at the bottom-left with y increasing upwards, the function flips the y-axis accordingly. Finally, it rounds the coordinates to the nearest integer pixel indices and clips them to ensure they lie within the image bounds. The output is a $2 \times N$ matrix of integer pixel coordinates suitable for image rasterization.

Detailed code for this function can be found in the rasterize_func.py file.

RESULTS

The output of the project consists of eight rendered images that are saved directly in the main project directory by running the `demo.py` script. These images are generated using two different shading models—Gouraud and Phong—and four lighting configurations: ambient-only, diffuse-only, specular-only, and full lighting. Each image filename follows the format `render_<shader>_<lighting>.png`, where `<shader>` is either `gouraud` or `phong`, and `<lighting>` corresponds to one of the four lighting modes. To generate the outputs, the user simply executes `python3 demo.py`, which handles the rendering process and saves the resulting images.

Shown below are two indicative demo category results: one when using gouraud shading and the other when using phong shading:

DEMO WITH SHADE_GOURAUD

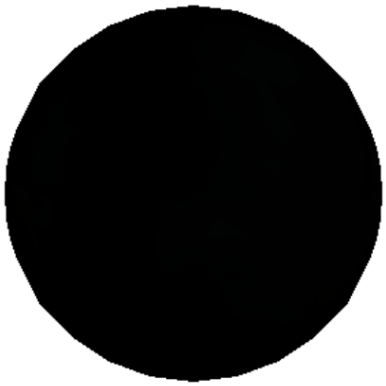

Fig. 4: demo with shade_gouraud (ambient)

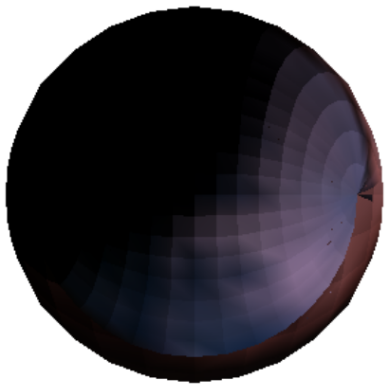

Fig. 5: demo with shade_gouraud (diffuse)

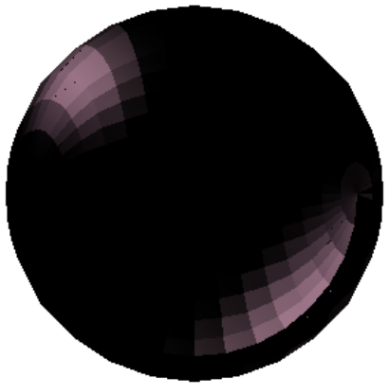

Fig. 6: demo with shade_gouraud (specular)

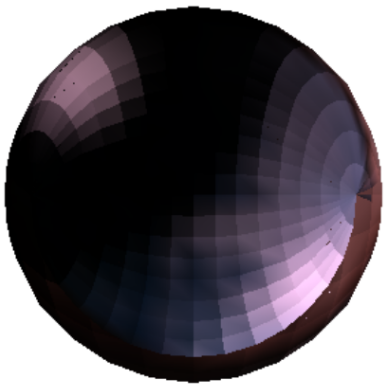

Fig. 7: demo with shade_gouraud (full)

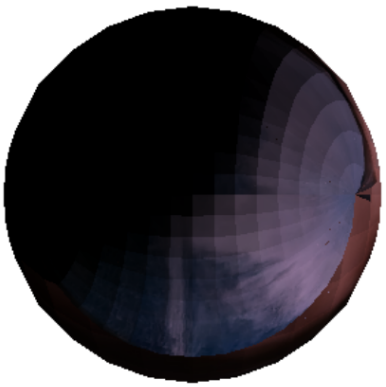

Fig. 9: demo with shade_phong (diffuse)

DEMO WITH SHADE_PHONG

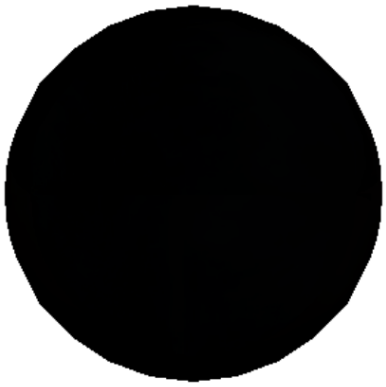

Fig. 8: demo with shade_phong (ambient)

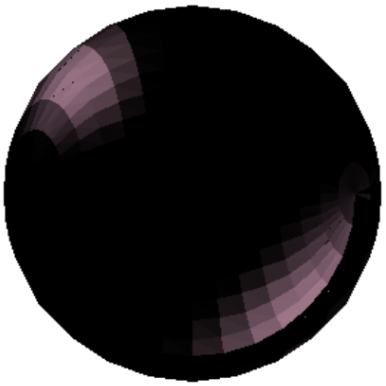

Fig. 10: demo with shade_phong (specular)

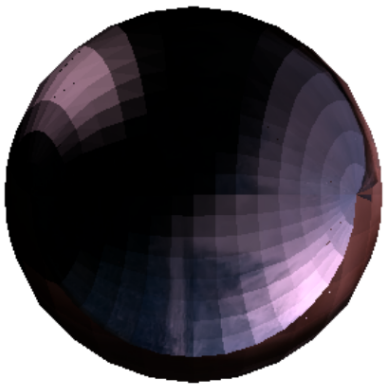

Fig. 11: demo with shade_phong (full)

OBSERVATIONS AND EVALUATION

This section presents a comprehensive analysis of rendered results using two classical shading techniques: **Gouraud shading** and **Phong shading**. For both methods, four rendering passes were performed: *ambient*, *diffuse*, *specular*, and *full* (combined). The key difference lies in the interpolation strategy used during shading:

The main difference between the two shading methods is:

- **Gouraud shading** calculates the lighting at each vertex and interpolates the resulting color across the triangles.
- **Phong shading** interpolates the normals across the triangles and calculates lighting per pixel.

***Ambient Lighting**:* Both Gouraud and Phong ambient renders appear completely black. This is expected since the ambient light is set to a very low value, and it is constant across the surface. There is no difference between the two methods in this case.

***Diffuse Lighting**:* The diffuse images look almost the same for both shading methods. In theory, Phong shading should give a smoother result, especially on curved surfaces. However, in this case, the difference is minimal. This could be due to the mesh being dense enough or the lighting setup being simple. As a result, both Gouraud and Phong produce very similar diffuse shading.

***Specular Lighting**:* The specular results also look nearly identical in both methods. Normally, Phong shading would give better highlights because it calculates them per pixel, while Gouraud can miss them if they don't fall near a vertex. But here, the highlight is smooth and present in both cases. That means the mesh and lighting were good enough that even Gouraud captured the specular effect well.

***Full Lighting**:* The full lighting images — which include ambient, diffuse, and specular — are also very similar in both shading methods. Any differences are subtle. The lighting appears smooth, and the highlights are clearly visible in both images.

CONCLUSION

In this specific setup, both Gouraud and Phong shading give very similar visual results across all lighting components. This suggests that:

- The mesh had enough resolution to avoid Gouraud artifacts.
- The light position and material properties did not create sharp highlights or gradients.

Normally, Phong shading is expected to give smoother results and more accurate highlights, especially on low-resolution meshes. But in this case, the differences are minimal, and both methods work well. Gouraud has the advantage of being faster, so in similar setups it could be used without visible loss in quality.