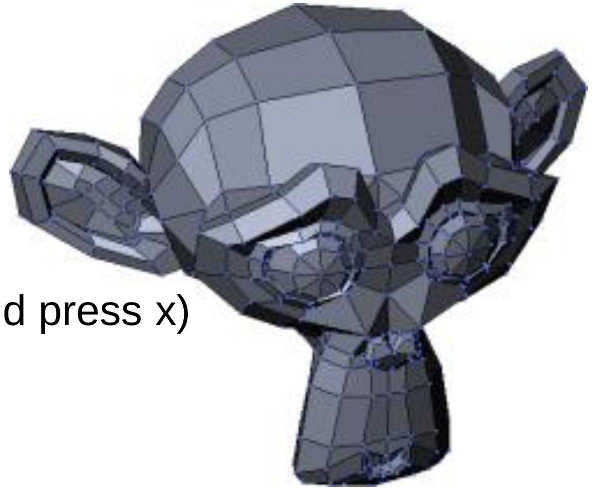
OpenGL
III -Basic Shaders

Stefan BORNHOFEN
EISTI

Load a Model (*.obj)

"Suzanne" is a well-known Blender test model.

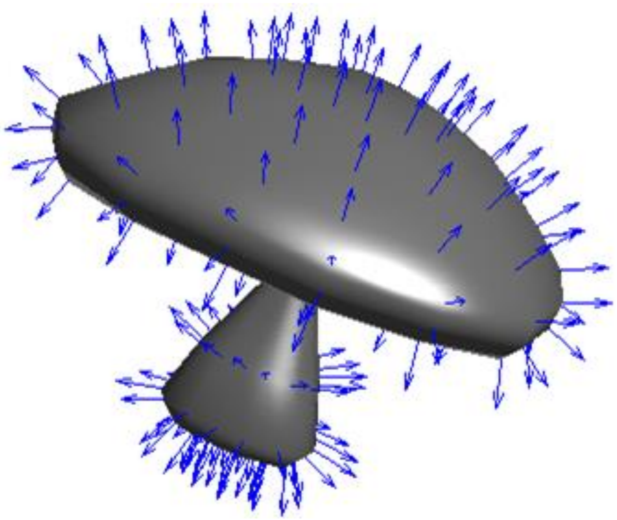


- Run Blender
- Remove all elements from the scene (right click on them and press x)
- Add > Mesh > Monkey
- Type n to display the Transform panel and
  - set the location to (0, 0, 0)
  - set the rotation to (0, 0, 0)
- Add a texture with UV mapping using "Smart UV Project"
- File > Export > Wavefront (.obj)
- To preserve the Blender orientation, set the following options:
  - Forward: -Z Forward
  - Up: Y Up
- Tick "Write Normals" and "Include UVs"
- Tick "Triangulate Faces" so that we get triangle faces instead of quad faces
- Blender will create two files, suzanne.obj and suzanne.mtl:
  - the .obj file contains the mesh : vertices and faces
  - the .mtl file contains information on materials (Material Template Library)
- We only use the mesh file.

```
#include <fstream>
#include <sstream>
#include <iostream>
#include <string>

void load_obj(const char* filename, vector<GLfloat> &mesh_data) {
          vector<glm::vec4> v;
          vector<glm::vec2> vt;
          vector<glm::vec3> vn;
          vector<vector<GLushort>> f;

          // 1) read file data into v, vt, vn and f
          ifstream in(filename, ios::in);
          if (!in)  { cerr << "Cannot open " << filename << endl; exit(1); }
          string line;
          while (getline(in, line))  {
                    if (line.substr(0, 2) == "v ") {
                              istringstream s(line.substr(2));
                              glm::vec4 v4; s >> v4.x; s >> v4.y; s >> v4.z, v4.w = 1.0f;
                              v.push_back(v4);
                    } ...
          }

          // 2) for each face f, store into mesh_data three consecutive vertices in the form:
          // v.x, v.y, v.z, v.w, vt.u, vt.v, vn.x, vn.y, vn.z
          ...
}
```

Normal Matrix

- The vertex shader needs to rotate the normals according to the model view MV
- However, we cannot directly apply the rotational part of MV because it might contain a scale effect, and normals must not be scaled
- The correct way consists in calculating NM as $NM = (MV^{-1})^T$



```
MV  = View * Model;
MVP = Projection * MV;
NM  = glm::transpose(glm::inverse(glm::mat3(MV)));
```

Suzanne vertex shader

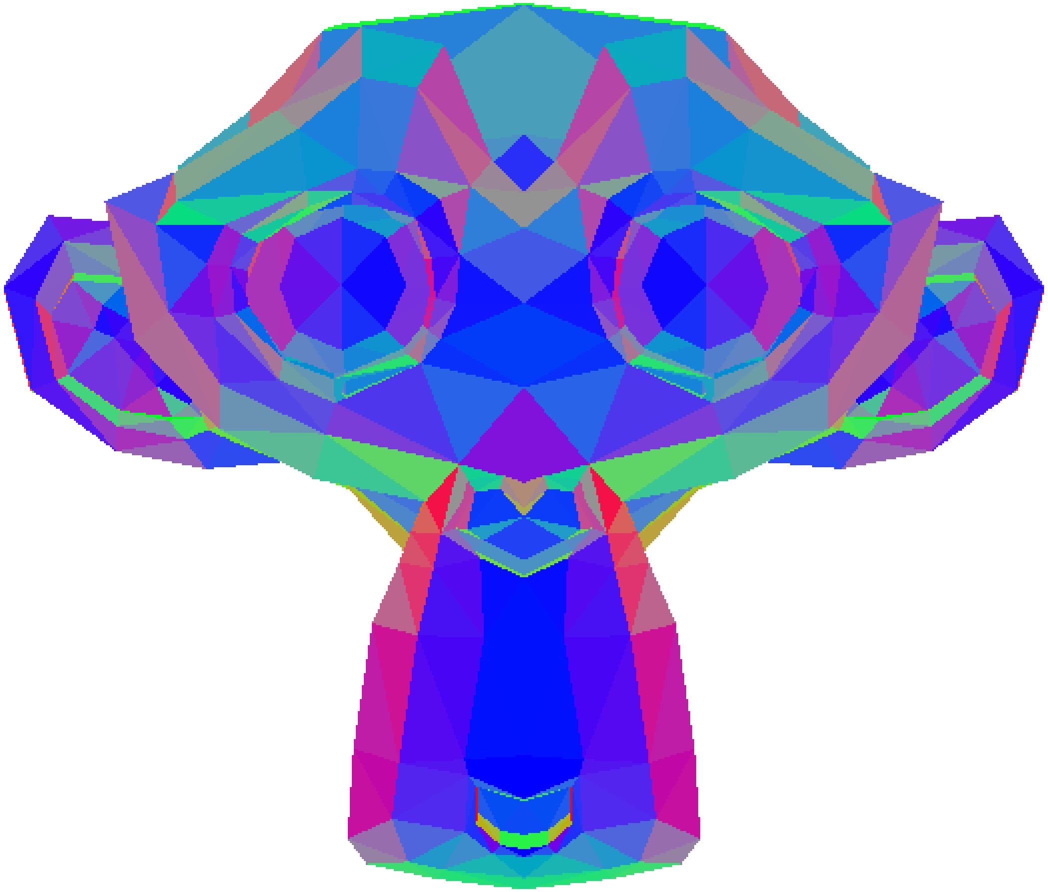
```
#version 430
in vec4 v_coord;
in vec2 v_texcoord;
in vec3 v_normal;

uniform mat4 MVP;
uniform mat3 NM; // Normal Matrix

out vec4 color;
out vec2 texcoord;

void main()
{
    // pass texcoord to fragment shader (not used for the moment)
    texcoord = v_texcoord;

    // display normals
    vec3 N = normalize(NM * v_normal);
    color = vec4(abs(N),1.0f);

    gl_Position = MVP*v_coord;
}
```

Suzanne fragment shader

```
#version 430

in vec4 color;
in vec2 texcoord;

out vec4 fColor; // final fragment color

void main()
{
    fColor = color;
}
```

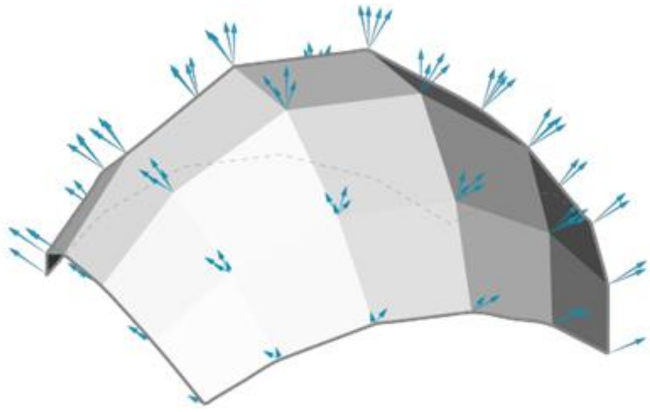
```
// load mesh
load_obj("suzanne.obj", suzanne_mesh_data);

[...]

// create vba with one vbo containing suzanne_mesh_data
glGenVertexArrays(1, vaoSuzanne); glBindVertexArray(vaoSuzanne);
glGenBuffers(1, &vbo_mesh_data); glBindBuffer(GL_ARRAY_BUFFER, vbo_mesh_data);
glBufferData(GL_ARRAY_BUFFER, suzanne_mesh_data.size() * sizeof(GLfloat), &suzanne_mesh_data[0],
GL_STATIC_DRAW);

[...]

// shader plumbing
GLuint attribute;
attribute  = glGetAttribLocation(shader, "v_coord"); glEnableVertexAttribArray(attribute);
glVertexAttribPointer(attribute, 4, GL_FLOAT, GL_FALSE, 9*sizeof(GLfloat), (GLvoid*)0);

attribute = glGetAttribLocation(shader, "v_texcoord"); glEnableVertexAttribArray(attribute);
glVertexAttribPointer(attribute, 2, GL_FLOAT, GL_FALSE, 9*sizeof(GLfloat), (GLvoid*)(4*sizeof(GLfloat)));

attribute = glGetAttribLocation(shader, "v_normal"); glEnableVertexAttribArray(attribute);
glVertexAttribPointer(attribute, 3, GL_FLOAT, GL_FALSE, 9*sizeof(GLfloat), (GLvoid*)(6*sizeof(GLfloat)));

[...]

// render
glDrawArrays(GL_TRIANGLES, 0, suzanne_mesh_data.size()/9);
```

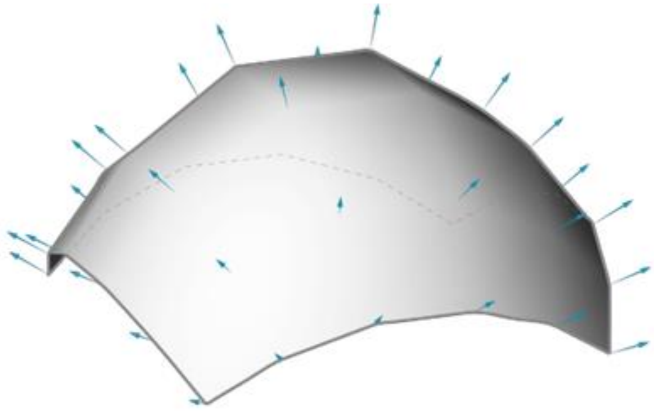
Hello Suzanne



Vertex normals

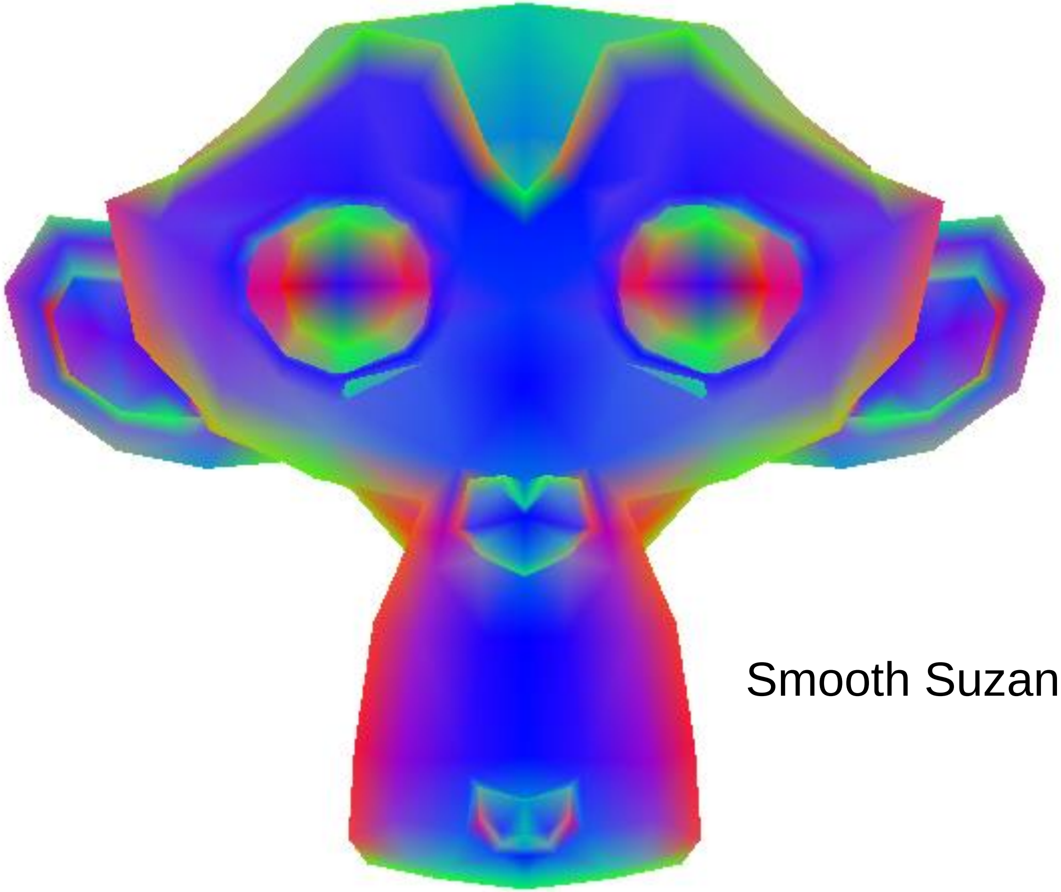
- If a vertex has multiple adjacent faces, the vertex normal is calculated by taking the average of the faces.
- Vertex normals are important for smooth visualization of meshes.





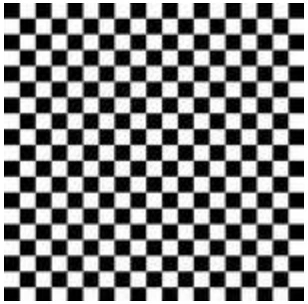
Exercise

1. **For each vertex, calculate the appropriate vertex normal.**
2. **Create a second VAO/VBO with the same format**
   `v.x, v.y, v.z, v.w, vt.u, vt.v, vn.x, vn.y, vn.z`
   where each vertex has vertex normals instead of face normals.



Smooth Suzanne

Textures



- Create a texture
- Bind the texture
- Change the fragment shader to

```
#version 430

in vec4 color; // ignored
in vec2 texcoord; // the interpolated UV coordinates
uniform sampler2D tex; // the currently bound texture

out vec4 fColor; // final fragment color

void main()
{
	fColor = texture(tex, texcoord);
}
```

Exercise

Textured Suzanne

```
fColor = texture(tex, texcoord);
```

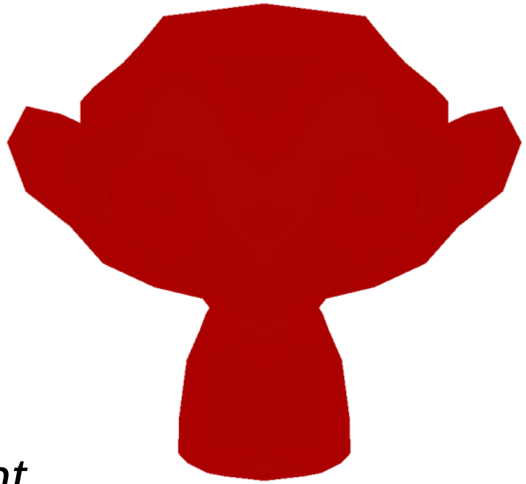
```
fColor = texture(tex, texcoord) * color;
```

Reflectance vs. Shading

- **Reflectance Model**: Gives an **intensity** at a point based on the light vector, normal vector, and other factors. Reflectance model determines **how light moves** *(Lambert, Phong, Blinn)*
- **Shading Model**: Determines how to **interpolate** across polygonal surfaces to make them look smooth (*Flat, Gouraud, Phong shading)*

Reflectance: Ambient Lighting

- A minimum brightness, even if there is no light hitting a surface directly
- Does not depend on the light source position
- Intensity is the same at all points



*ambient = Ka * lightColor*

- *Ka is the material's ambient reflectance*
- *lightColor is the color of the incoming ambient light.*

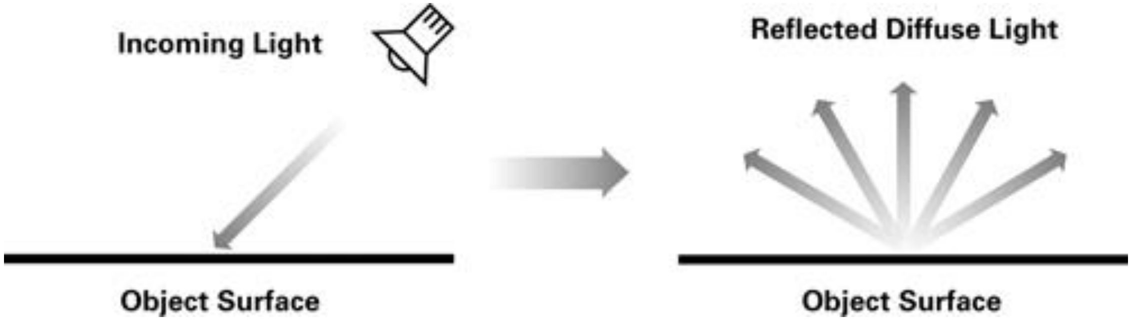
Exercise: Write a vertex shader calculating ambient lighting

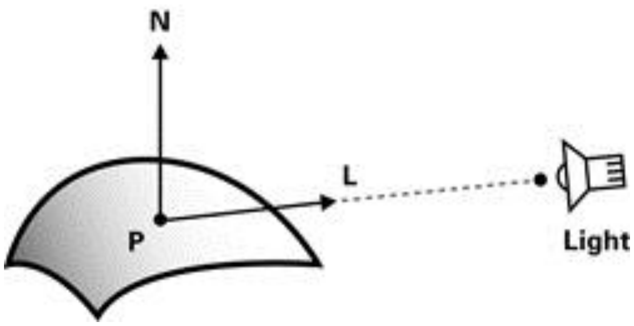
Reflectance: Diffuse Lighting

- Diffuse reflection scatters light equally in all directions (“Lambertian surface”)
- Intensity depends on the angle of incoming light

$$diffuse = Kd * lightColor * max(N \cdot L, 0)$$

- *Kd is the material's diffuse color,*
- *lightColor is the color of the incoming diffuse light*
- *N is the normalized surface normal,*
- *L is the normalized vector toward the light source*
- *P is the point being shaded*





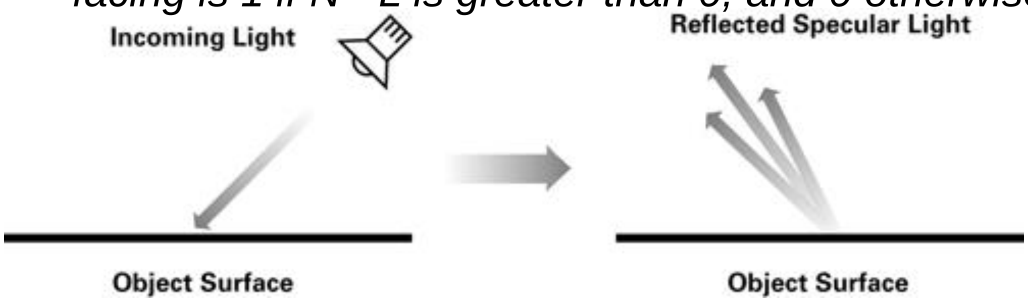
Exercise: Write a vertex shader calculating diffuse lighting

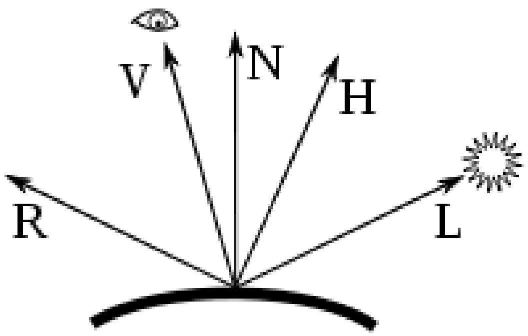
Reflectance: Specular Lighting

- Represents light scattered predominantly around the mirror direction.
- Intensity depends on the angle between the surface normal and the halfway vector

*specular = Ks * lightColor * facing * (max(R · V, 0) ^shininess)*

- *Ks is the material's specular color,*
- *lightColor is the color of the incoming specular light,*
- *N is the normalized surface normal,*
- *V is the normalized vector toward the viewpoint,*
- *L is the normalized vector toward the light source,*
- *R is the perfectly reflected light beam*
- *H is the normalized vector that is halfway between V and L,*
- *P is the point being shaded*
- *facing is 1 if N · L is greater than 0, and 0 otherwise.*





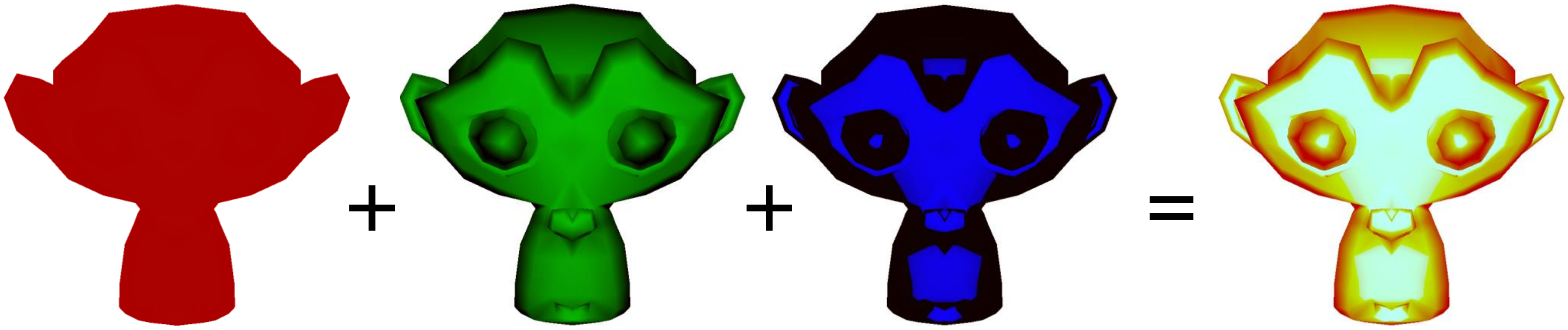
Exercise: Write a vertex shader calculating specular lighting

Reflectance: Phong Lighting

Reflection = Ambient + Diffuse + Specular

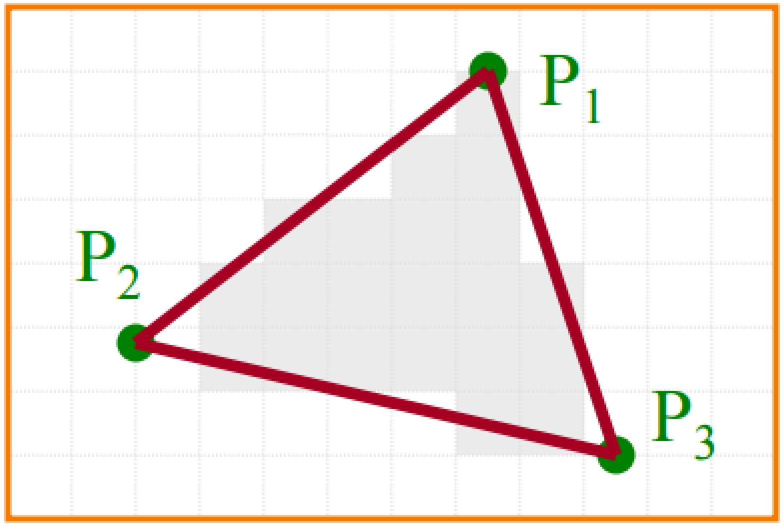
- Looks adequate
- Cheap to compute
- Intuitive parameters that can be tweaked to control appearance.

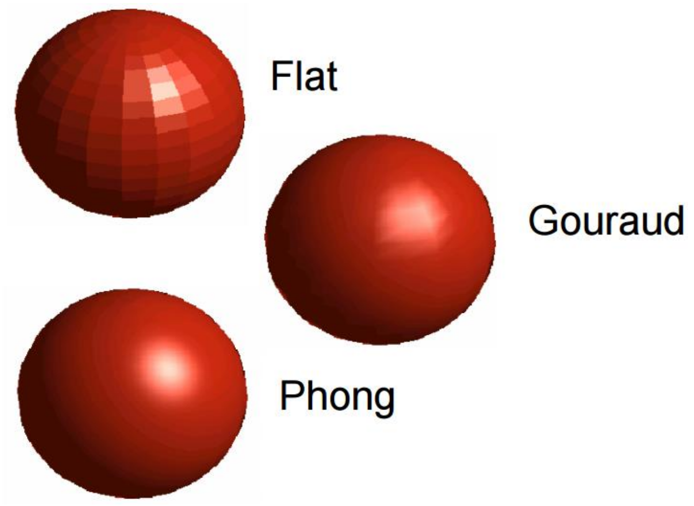
- Works well for only a limited set of materials.
- A plastic or rubbery appearance is the most common result.



Shading techniques

After rasterization, determine a color for each filled pixel.

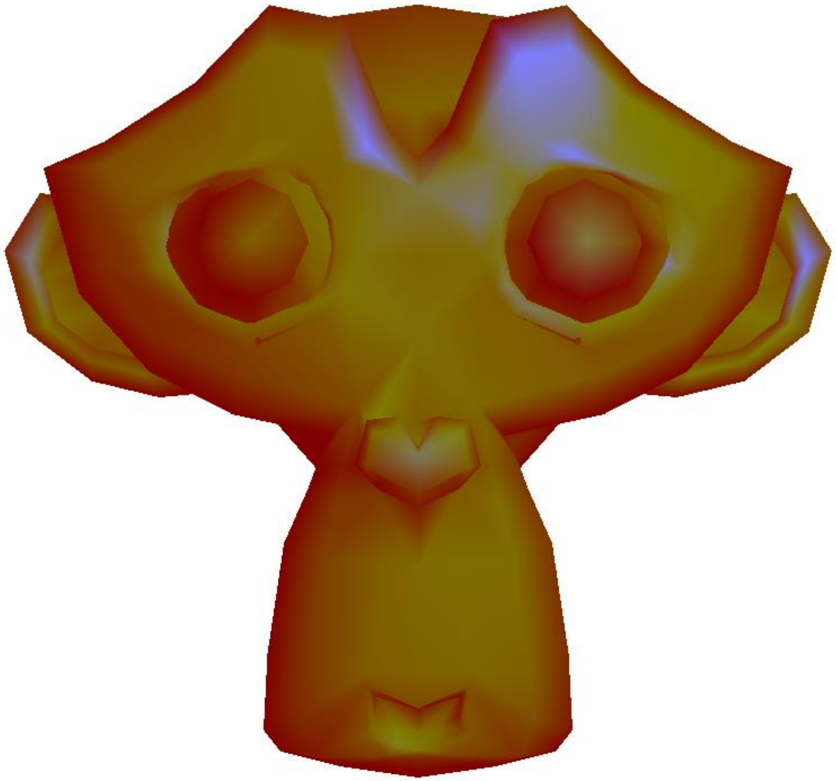




- Flat Shading: per-polygon lighting
- Gouraud Shading: per-vertex lighting
- Phong Shading: per-pixel lighting

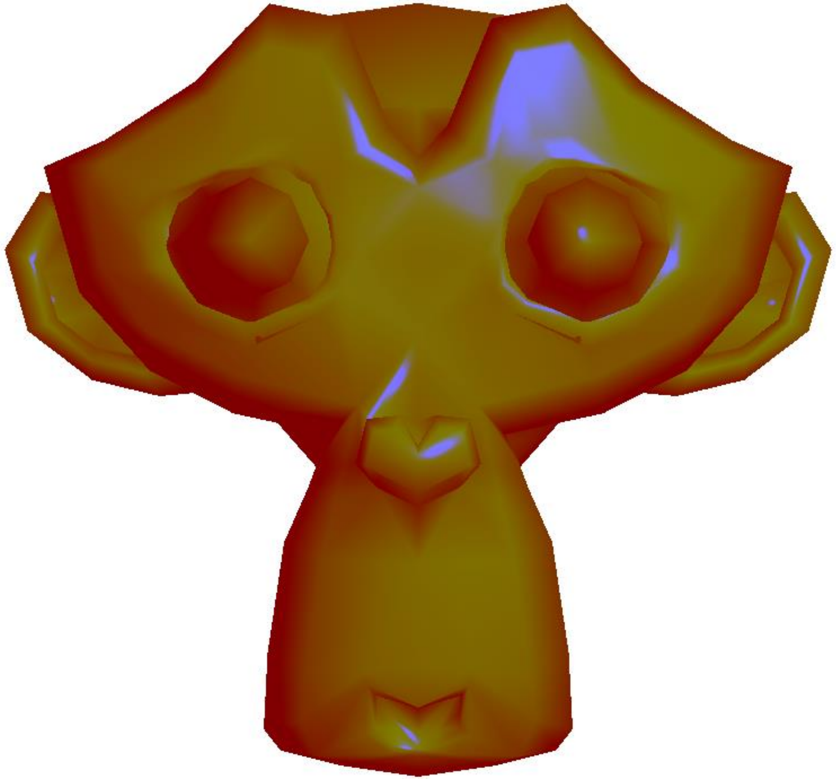
Gouraud Shading

- The vertex shader computes the normal and applies the illumination model to the vertex
- The fragment shader just interpolates the vertex intensities over the surface polygon



Phong Shading

- The <u>vertex shader</u> just computes the normal and passes the result to the fragment shader
- The <u>fragment shader</u> interpolates the normals over the surface polygon and applies the illumination model to each surface point



Exercise

Implement Gouraud and Phong shading. The user can interactively change ambient, diffuse, specular terms, shininess and light position.