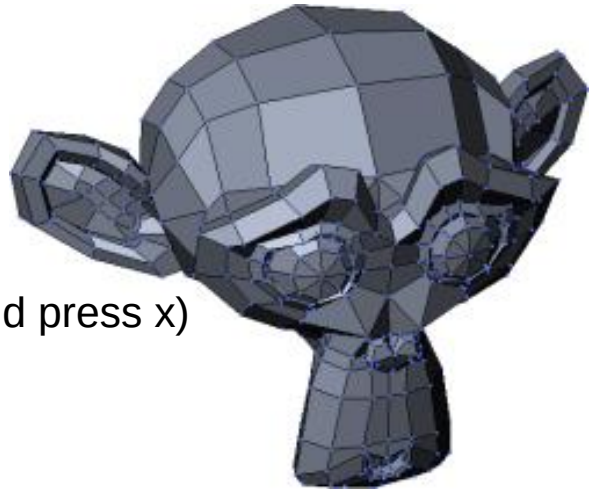
OpenGL
III –Basic Shaders

Stefan BORNHOFEN

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
        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 ") { // vertex data
                        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);
                }
                ... do something similar for vt, vn, and f
        }

        // 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
        // Caution: obj lists are indexed starting with 1, your vectors with 0 !
        ...
}
```

Normal color vertex shader

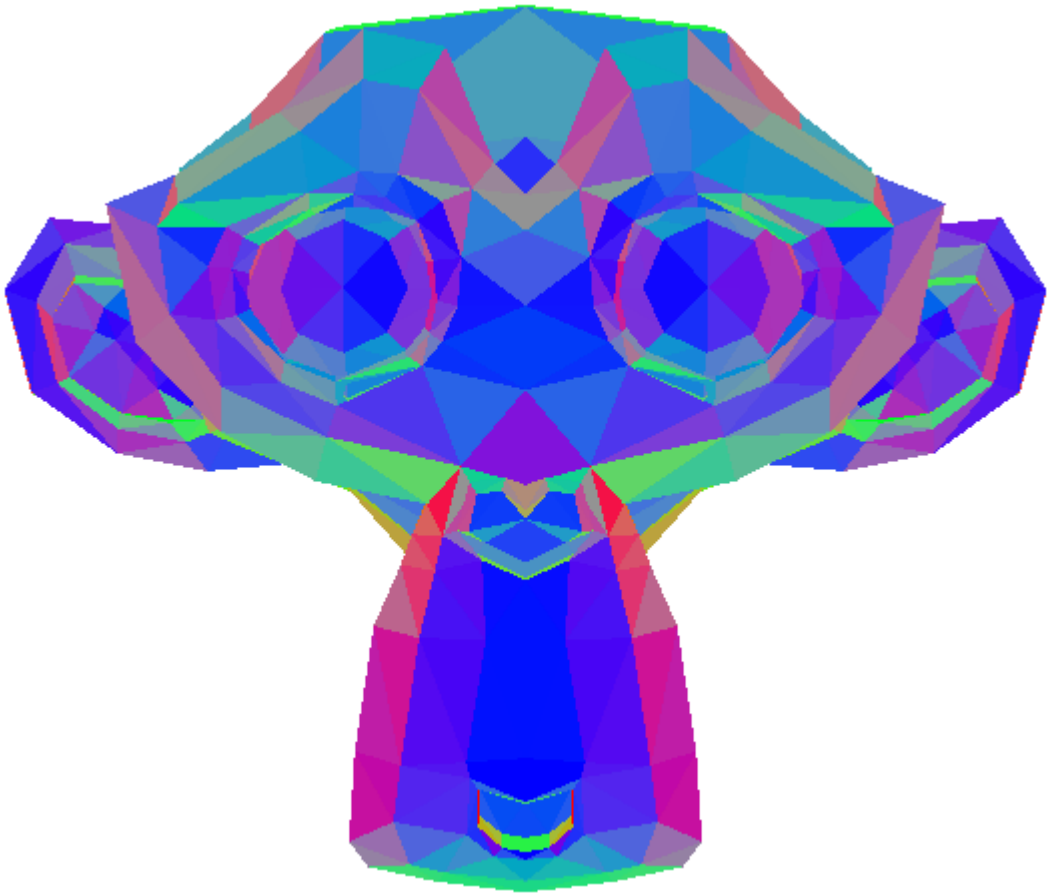
```
#version 430
in vec4 v_coord;
in vec2 v_texcoord;
in vec3 v_normal;

uniform mat4 MVP;
out vec4 color;
out vec2 texcoord;

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

    // display normals
    color = vec4(abs(v_normal),1.0);

    gl_Position = MVP*v_coord;
}
```

Normal color fragment shader

```
#version 430

in vec4 color;
in vec2 texcoord;

out vec4 fColor; // final fragment color

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

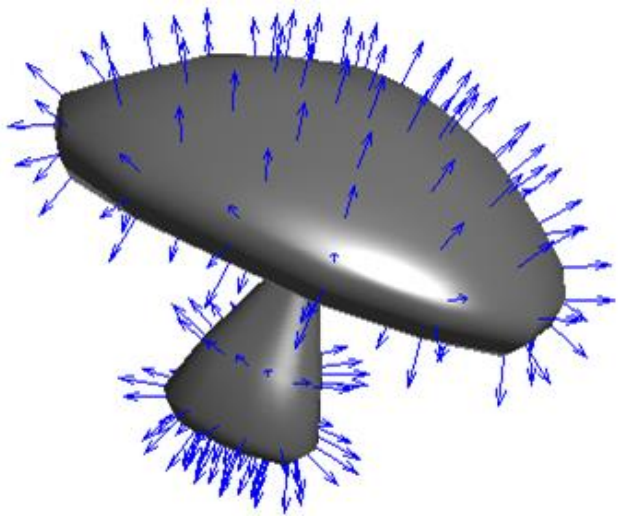
```
// load mesh
load_obj("suzanne.obj", suzanne_mesh_data);

[...]

// create vao 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



Normal Matrix

- The vertex shader needs to rotate the normals according to the model view MV
- We cannot directly use 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 $NM = (MV^{-1})^T$



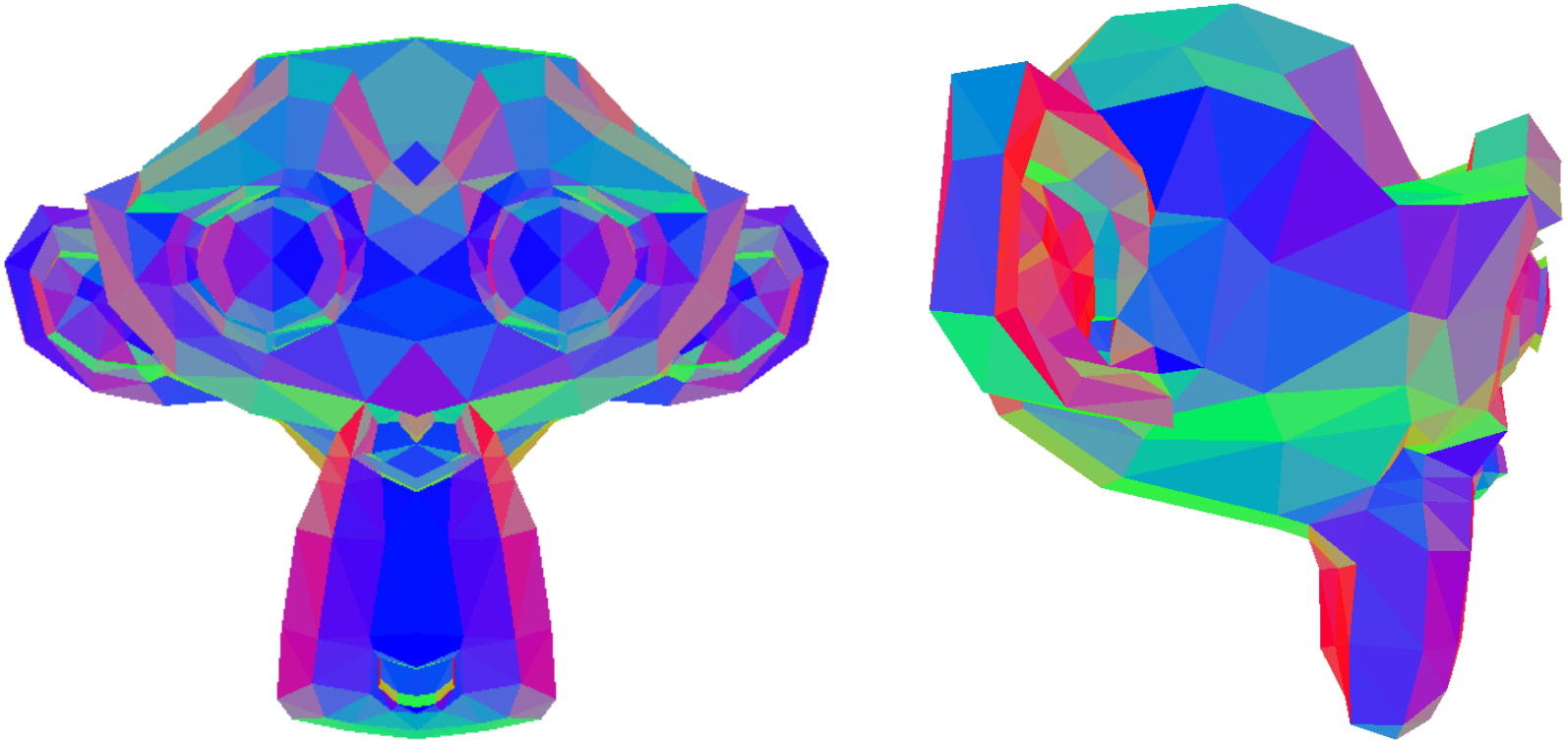
```
MV  = View * Model;
MVP = Projection * MV;
NM  = glm::transpose(glm::inverse(glm::mat3(MV)));
```

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;
}
```

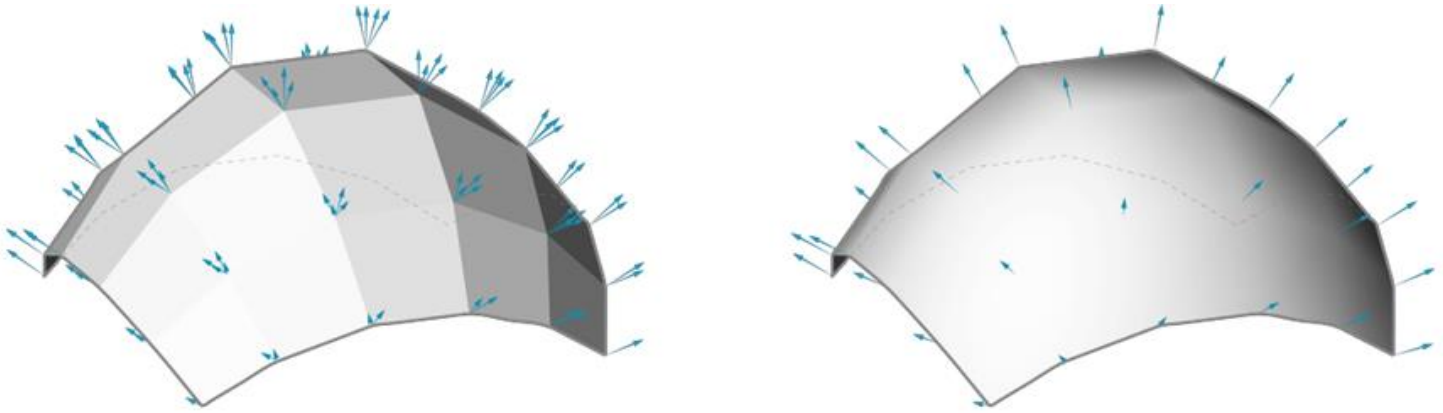
Exercise

Turn the normal vectors together with the model.



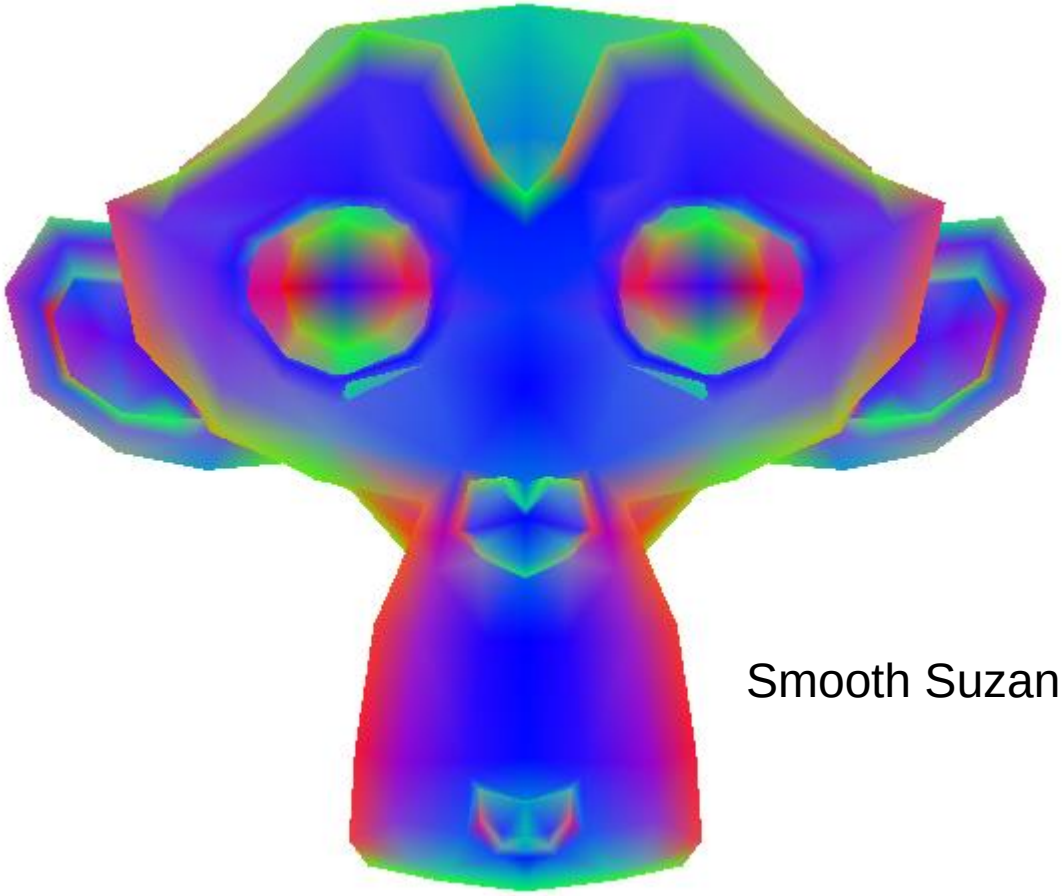
Vertex normals

- 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.

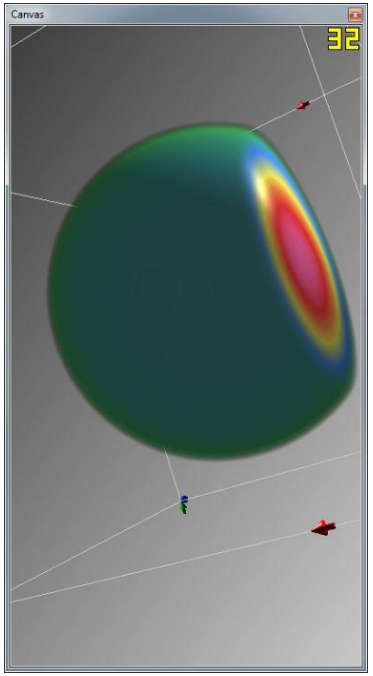


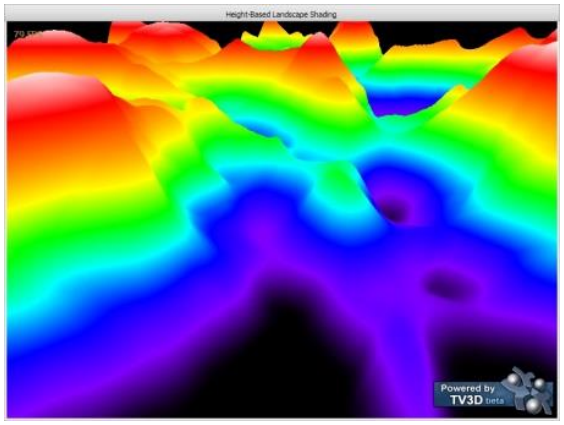
Smooth Suzanne

Texturing

- Apply a 1D, 2D, or 3D image to geometric primitives
- Uses of Texturing
  - simulating materials
  - reducing geometric complexity (often the surface of an object is viewed only from a distance)

Visual detail in the image,
not in the geometry!





Texture Objects

Stored on the GPU

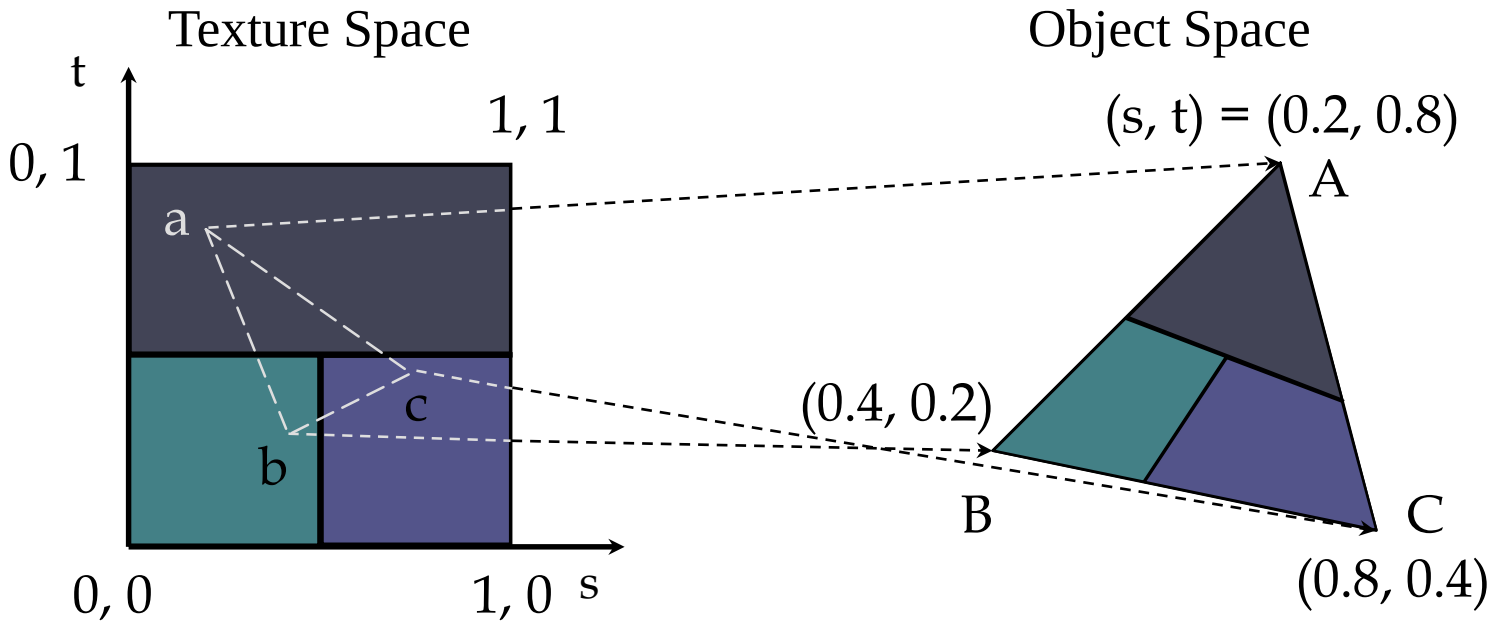
One image per texture object

May be reused by several graphics objects

- Generate texture name
```
glGenTextures (1, *texId);
```
- Bind textures before using
```
glBindTexture (target, texId);
```
- Copy a texture image from CPU memory
```
glTexImage2D (target, level, components,
  w, h, border, format, type, *texels);
```
Dimensions w, h should be powers of 2 (if not, use gluScaleImage)
- Delete textures when you are done
```
glDeleteTextures(1, *texId);
```

Texture Mapping

Based on texture coordinates specified at each vertex.

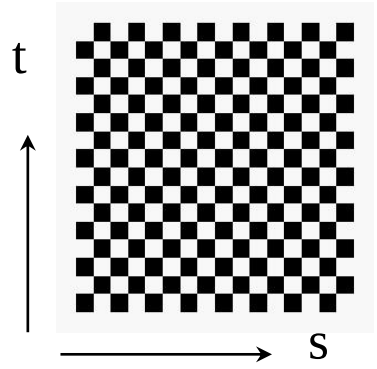


- Wrap Modes
  - Clamping or repeating
- Filter Modes
  - Minification and magnification

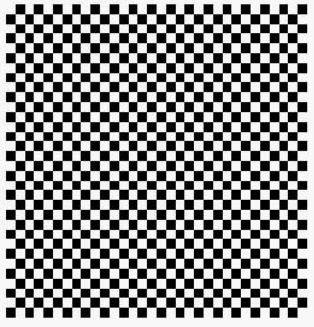
Wrap Mode

Determines what should happen if a texture coordinate lies outside of the [0,1] range.

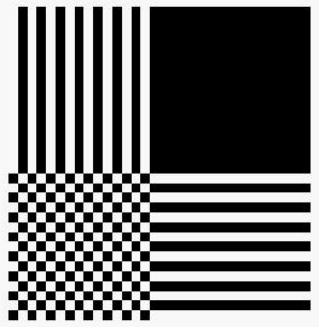
```
glTexParameteri (GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameteri (GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_T, GL_REPEAT)
```



texture



GL_REPEAT wrapping



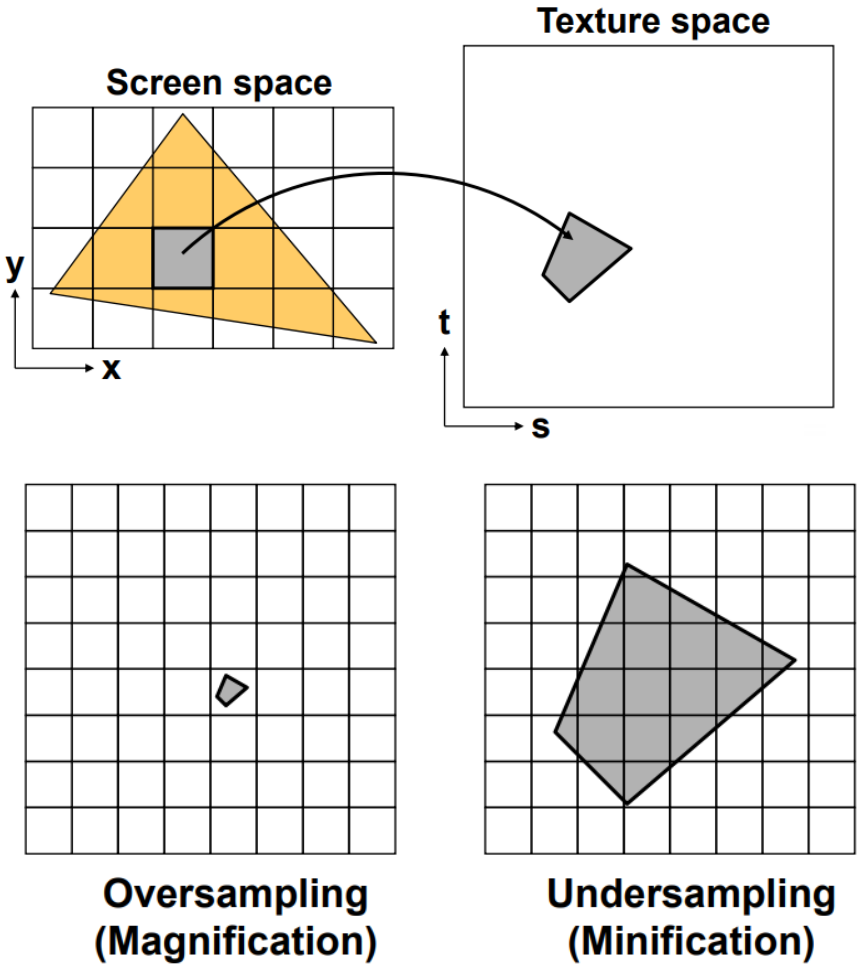
GL_CLAMP wrapping

Filter Modes

Sampling density in texture space rarely matches the sample density of the texture itself.

Texture mapping is subject to aliasing errors that can be controlled through filtering.

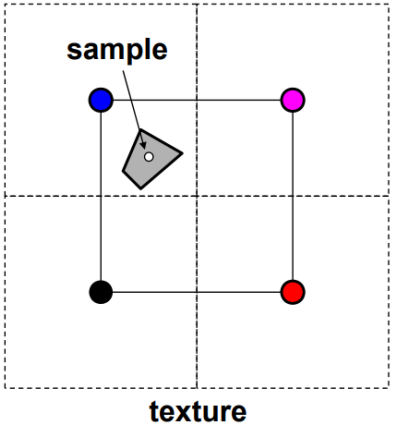
Filter modes control how pixels are minified or magnified.




```
glTexParameteri (target, type, mode);
```

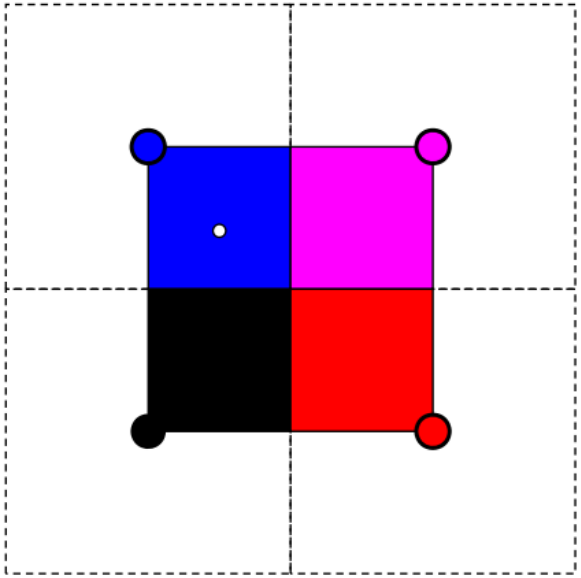
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, ...);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, ...);
```

Oversampling

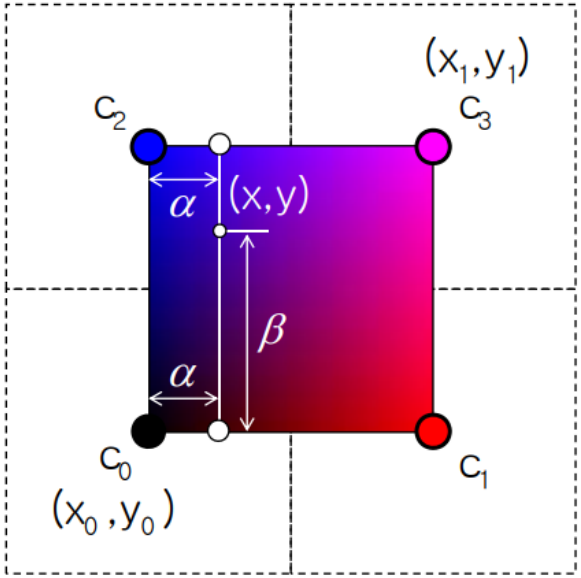


Many pixels map to an area contained by only one texture pixel.

- Nearest point sampling: use the texel closest to the point sample
- Linear filtering: use average of 2x2 group of texels close to the point sample



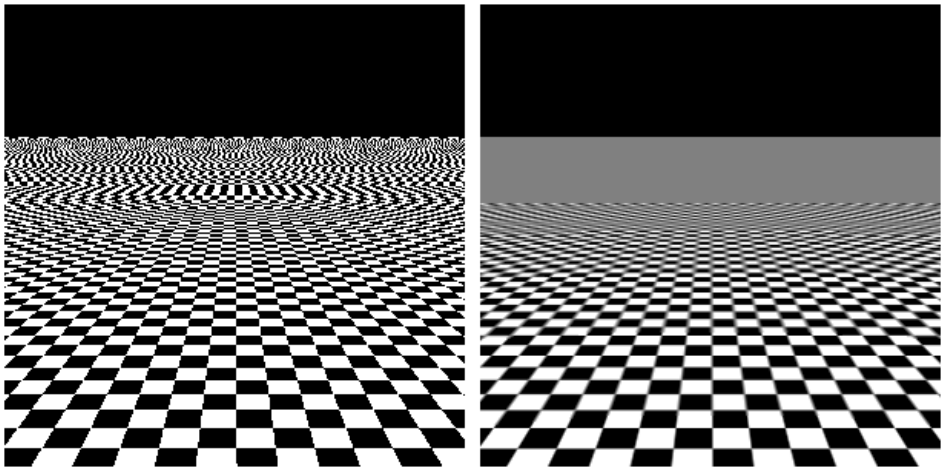
GL_NEAREST



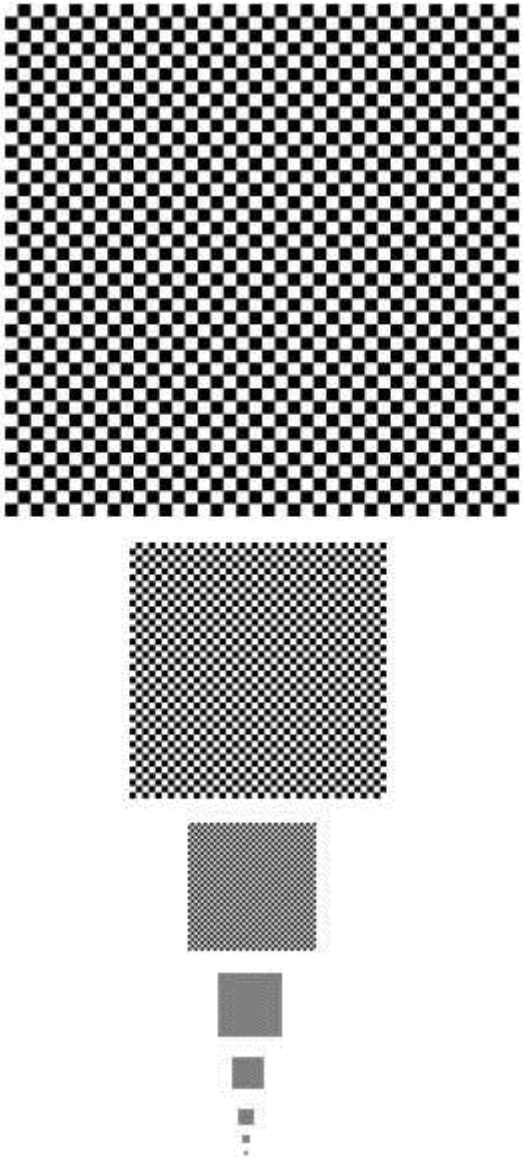
GL_LINEAR

Undersampling (Mipmapping)

- One pixel maps to an area covering many texture pixels
- If we skip pixels while reading a source image, we get undersampling artifacts (“shimmering”)
- A good way to solve this is to reduce the number of pixels to read from, by creating prefiltered texture maps of decreasing resolutions



No mipmapping *Mipmapping*



- `glGenerateMipmap(GL_TEXTURE_2D);`

Load Texture (from bmp)

```
GLuint loadTextureFromBMP(const char * filename, int width, int height) {
    GLuint texture = 0; unsigned char * data;
    FILE * file; errno_t err;
    if ((err = fopen_s(&file, filename, "rb")) != 0) printf("Error: Texture was not opened.\n");
    else {
          data = (unsigned char *)malloc(width * height * 3);
          fread(data, width * height * 3, 1, file);
          fclose(file);
          for (int i = 0; i < width * height; ++i) { // bmp files are encoded BGR and not RGB
                    int index = i * 3; unsigned char B, R;
                    B = data[index]; R = data[index + 2];
                    data[index] = R; data[index + 2] = B;
          }
          glGenTextures(1, &texture);
          glBindTexture(GL_TEXTURE_2D, texture);
          glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
          glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
          glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
          glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
          glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
          glGenerateMipmap(GL_TEXTURE_2D);
          free(data);
    }
    return texture;
}
```

Procedural Texture (example)

```
GLuint createTexureChecker() {
 const int CHECKERSIZE = 256;
 float *checkerImage = new float[CHECKERSIZE*CHECKERSIZE * 4];
 int i, j; float c;
 for (i = 0; i < CHECKERSIZE; i++)
  for (j = 0; j < CHECKERSIZE; j++) {
    c = (((i & 0x8) == 0) ^ ((j & 0x8)) == 0); // one square = 8 pixels
    checkerImage[4 * (i*CHECKERSIZE + j)]   = (GLfloat)c;
    checkerImage[4 * (i*CHECKERSIZE + j)+1] = (GLfloat)c;
    checkerImage[4 * (i*CHECKERSIZE + j)+2] = (GLfloat)c;
    checkerImage[4 * (i*CHECKERSIZE + j)+3] = (GLfloat)1.0f;
  }
 GLuint texture;
 glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, CHECKERSIZE, CHECKERSIZE, 0, GL_RGBA, GL_FLOAT, checkerImage);
 glGenerateMipmap(GL_TEXTURE_2D);
 delete[] checkerImage;
 return texture;
}
```

Using Textures in Shaders

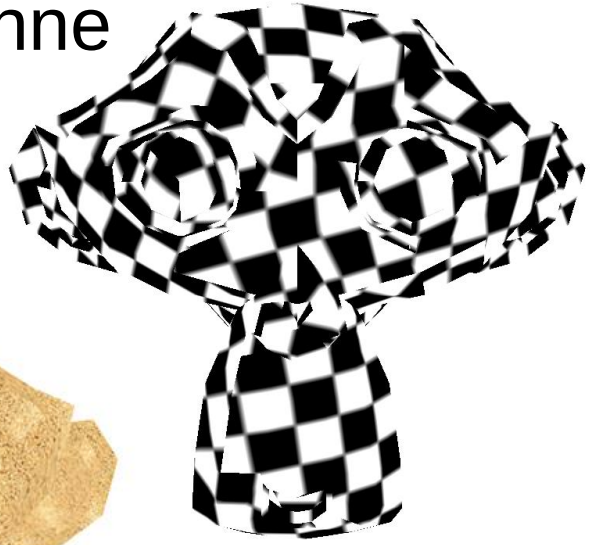
- The vertex shader receives the texture coordinates and passes them to the fragment shader

- The fragment shader has access to the texture and fetches the texture color. 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)*

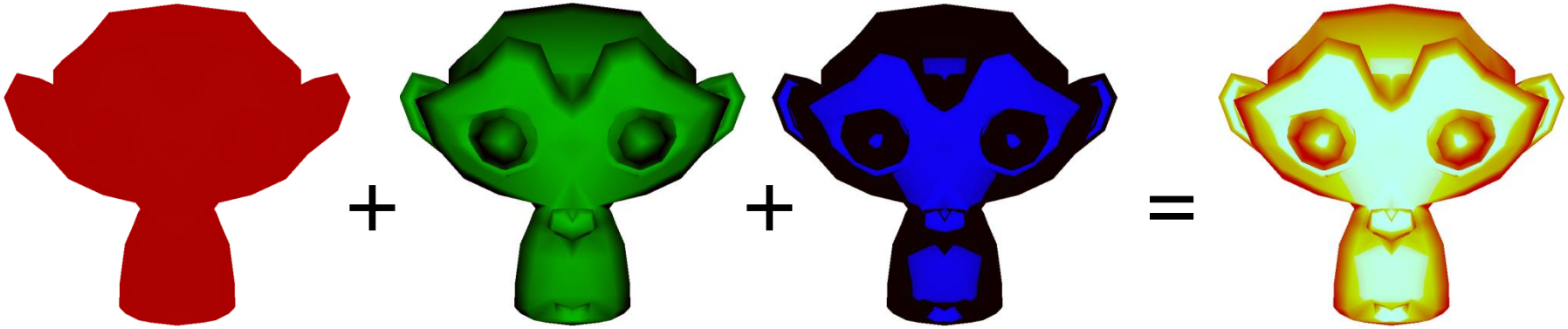
- **Shading Model**: Determines how to **interpolate** across polygonal surfaces to make them look smooth (*Flat, Gouraud, Phong shading)*

Reflectance: Phong Lighting

Reflection = Ambient + Diffuse + Specular

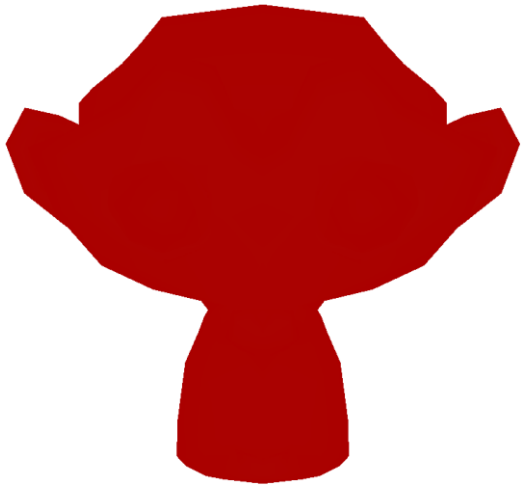
- Looks adequate
- Cheap to compute
- Intuitive parameters that can be tweaked to control appearance.

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



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

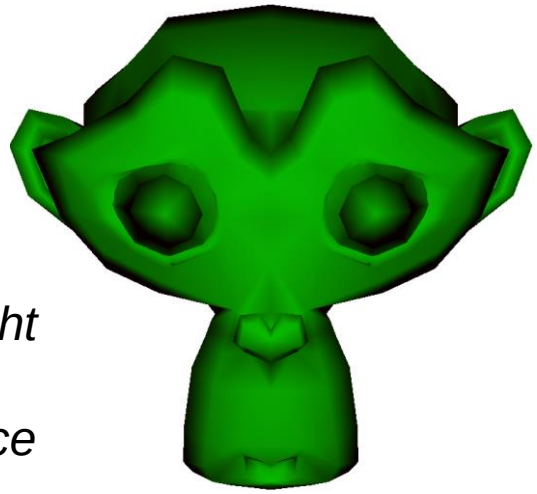


*ambient = Ka * lightColor*

- *vec4 Ka is the material's ambient reflectance*
- *vec4 lightColor is the color of the incoming ambient light.*

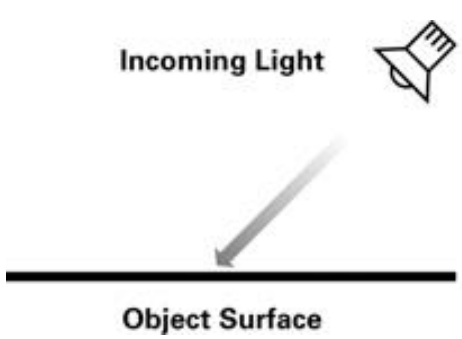
Reflectance: Diffuse Lighting

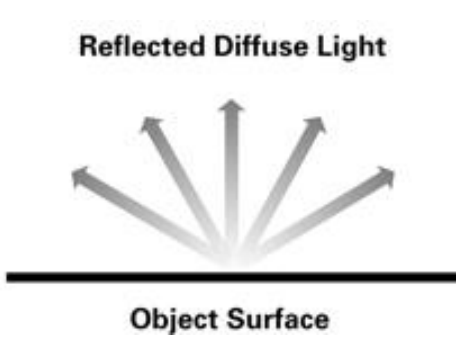
- Diffuse reflection scatters light equally in all directions (“Lambertian surface”)
- Intensity depends on the angle of incoming light

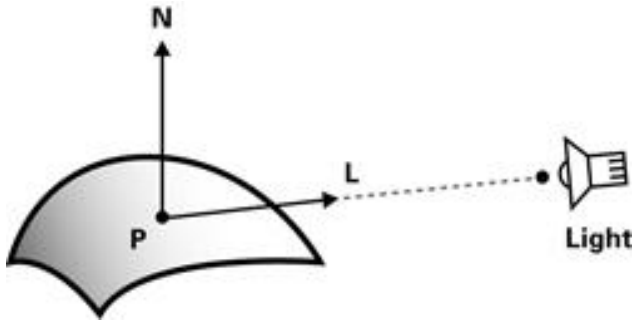


*diffuse = Kd * lightColor * max(N · L, 0)*

- *vec4 Kd is the material's diffuse color*
- *vec4 lightColor is the color of the incoming diffuse light*
- *vec3 N is the normalized surface normal*
- *vec3 L is the normalized vector toward the light source*
- *vec3 P is the point to be shaded*





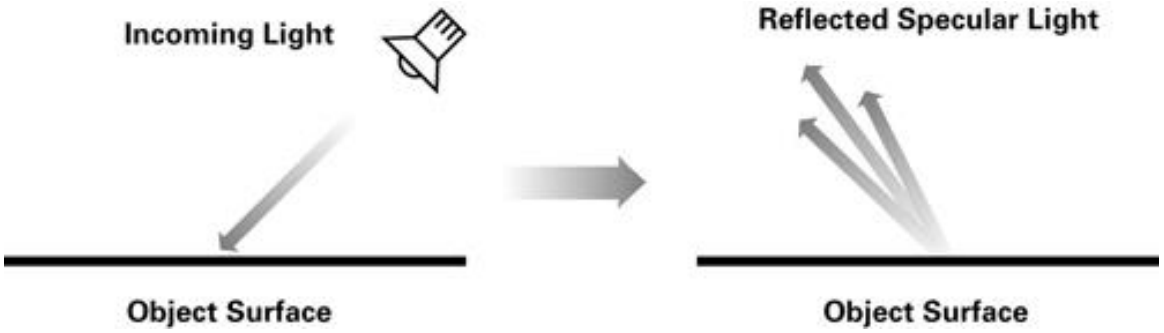


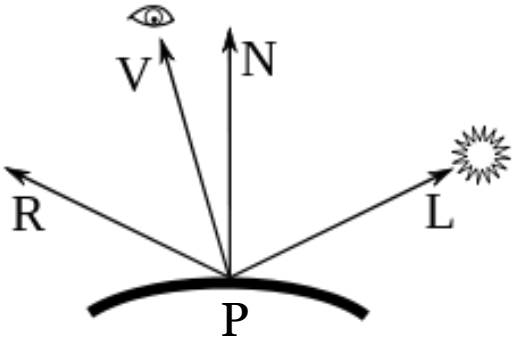
Reflectance: Specular Lighting

- 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)*

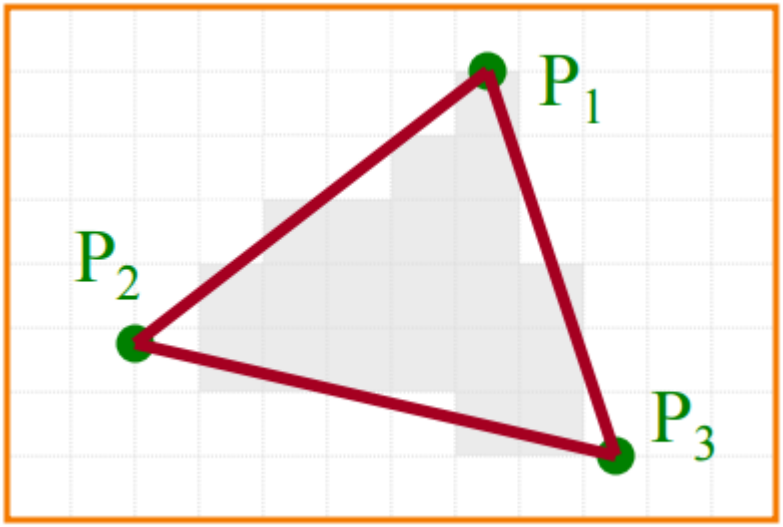
- *vec4 Ks is the material's specular color,*
- *vec4 lightColor is the color of the incoming specular light*
- *float shininess allows modulating the specular highlights*
- *vec3 N is the normalized surface normal,*
- *vec3 V is the normalized vector toward the viewpoint*
- *vec3 L is the normalized vector toward the light source*
- *vec3 R is the perfectly reflected light beam*
- *vec3 P is the point to be shaded*
- *float facing is 1 if N · L is greater than 0, and 0 otherwise*

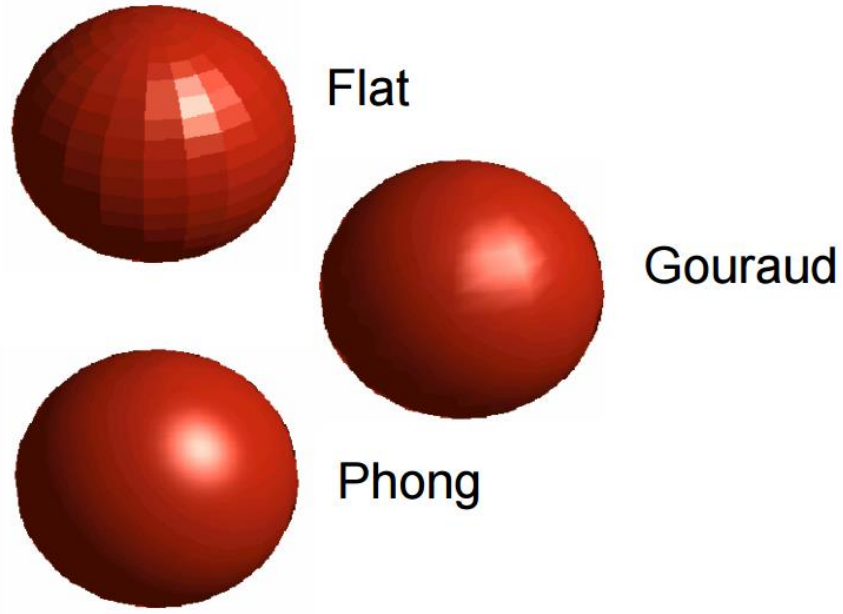




Shading techniques

After rasterization, determine a color for each filled pixel.





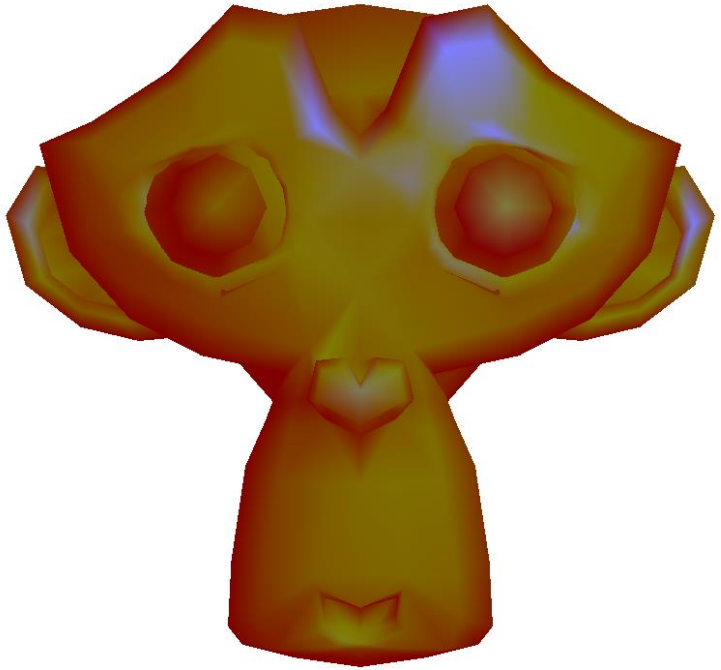
- Flat Shading: per-polygon lighting
- Gouraud Shading: per-vertex lighting
- Phong Shading: per-pixel lighting

Gouraud Shading

- The vertex shader computes

```
vec3 P = vec3(MV * v_coord); // the point being shaded
vec3 N = normalize(NM * v_normal) // the normal vector
```

  and applies the illumination model (A+D+S) to the vertex
- The fragment shader just interpolates the resulting color over the surface polygon



Gouraud Shading

Vertex shader

```
#version 430
in vec4 v_coord;
in vec3 v_normal;
uniform mat4 MV; // ModelView
uniform mat4 MVP; // Projection * ModelView
uniform mat3 NM; // Normal Matrix
uniform vec4 Ka;
uniform vec4 Kd;
uniform vec4 Ks;
uniform float shininess;
uniform vec3 lightpos;
out vec4 color;

void main()
{
   gl_Position = MVP*v_coord;
   // compute vec3: P, N, L, V, R
   ...
   // compute ambient term:
   vec4 amb = ...
   // compute diffuse term:
   vec4 diff = ...
   // compute specular term:
   vec4 spec = ...
   // compute total color:
   color = amb + diff + spec;
   color = clamp(color, 0.0, 1.0);
}
```

Fragment shader

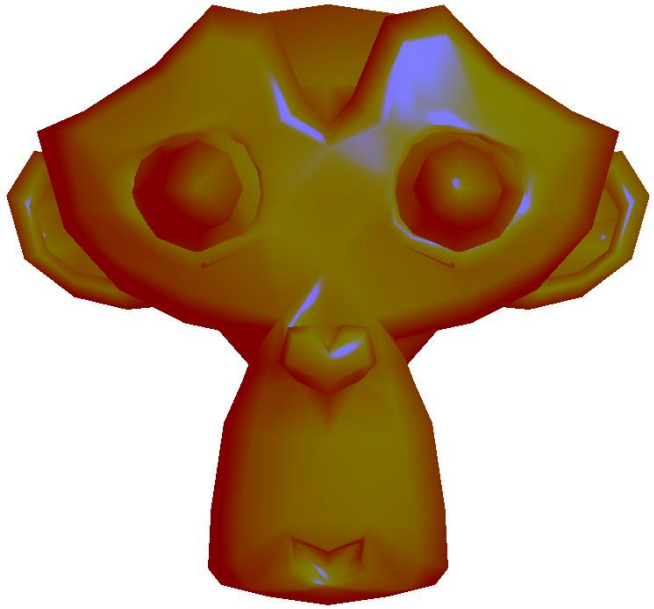
```
#version 430

in vec4 color;
out vec4 fColor

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

Phong Shading

- The vertex shader only computes

```
vec3 P = vec3(MV * v_coord); // the point being shaded
vec3 N = normalize(NM * v_normal) // the normal vector
```

  and passes these values to the fragment shader
- The fragment shader receives the interpolated values of P and N and applies the illumination model (A+D+S) to each surface point



Phong Shading

Vertex shader

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

uniform mat4 MV; // ModelView
uniform mat4 MVP; // Projection * ModelView
uniform mat3 NM; // Normal Matrix

out vec3 P;
out vec3 N;

void main()
{
   // compute P, N:
   P = ...
   N = ...
   gl_Position = MVP*v_coord;
}
```

Fragment shader

```
#version 430

in vec3 P;
in vec3 N;
uniform vec4 Ka;
uniform vec4 Kd;
uniform vec4 Ks;
uniform float shininess;
uniform vec3 lightpos;
uniform sampler2D tex;

out vec4 fColor;

void main()
{
   // compute L, V, R
   ...
   // compute ambient term:
   vec4 amb =...
   // compute diffuse term:
   vec4 diff =...
   // compute specular term:
   vec4 spec =...
   // compute total color:
   fColor = amb + diff + spec;
   fColor = clamp(fColor, 0.0, 1.0);
}
```

Exercise

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