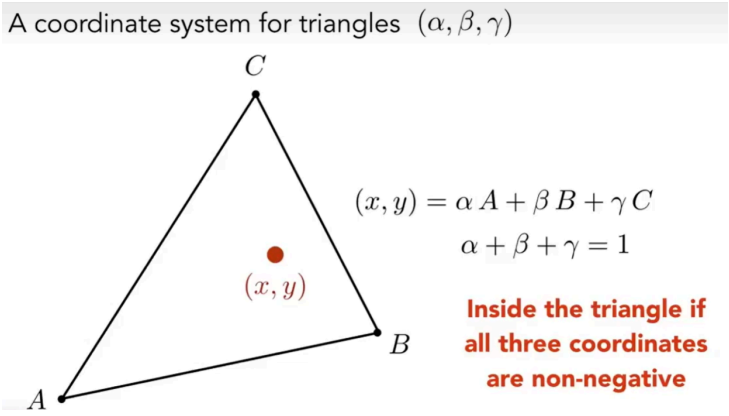
三角形插值

首先是重心坐标。



当 $\alpha+\beta+\gamma=1$ 时（x,y,z）在三角形所在平面内，当 α,β,γ 均非负时（x,y,z）在三角形内。（待证）

首先一个面的方程是 Qx+Wy+Ez+R=0，把 A,B,C 三个点带入进去得到 A,B,C 所在面的方程组：

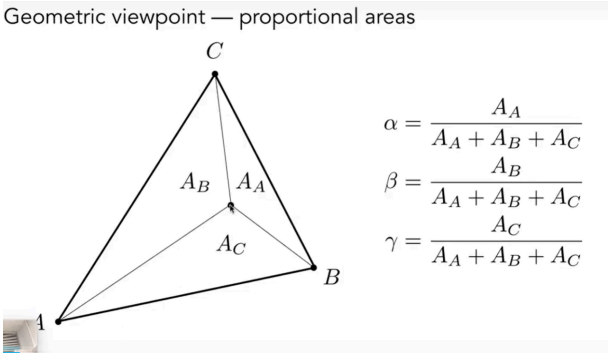
$$\begin{pmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \end{pmatrix} \begin{pmatrix} Q \\ W \\ E \\ R \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

这里我们默认 Q,W,E,R 已经解出来了，我们知道左边矩阵行向量的任意线性组合也满足这个方程组，其实就是：

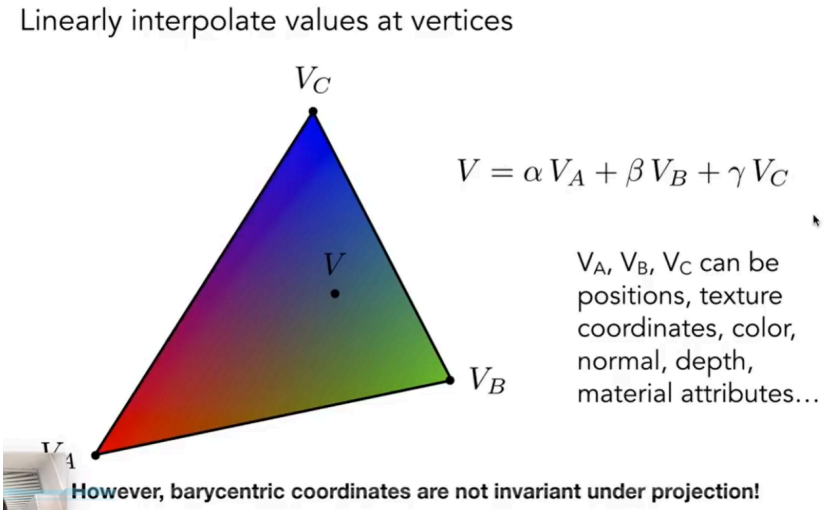
$$\begin{pmatrix} \alpha a_x + \beta b_x + \gamma c_x & \alpha a_y + \beta b_y + \gamma c_y & \alpha a_z + \beta b_z + \gamma c_z & \alpha + \beta + \gamma \end{pmatrix} \begin{pmatrix} Q \\ W \\ E \\ R \end{pmatrix} = (0)$$

但如果 $\alpha+\beta+\gamma \neq 1$ 的话就不是原先的面的方程了。

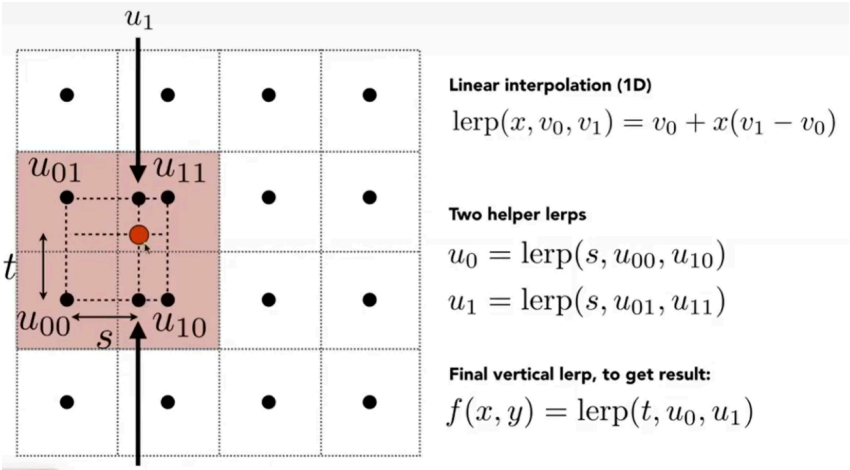
然后是重心坐标的另一种定义。



接下来就是应用重心坐标来插值。



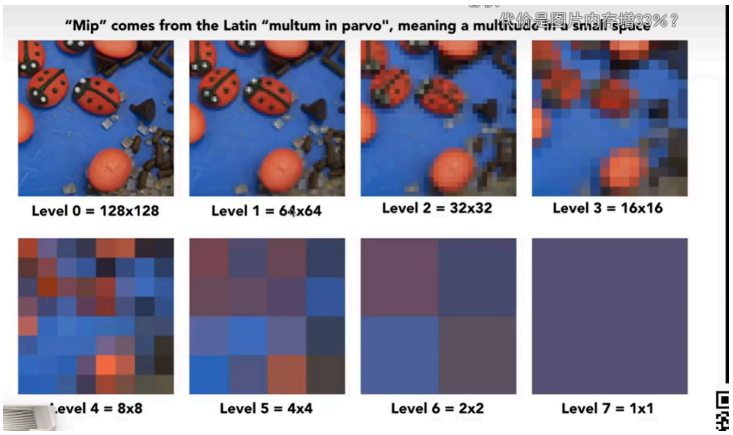
纹理插值



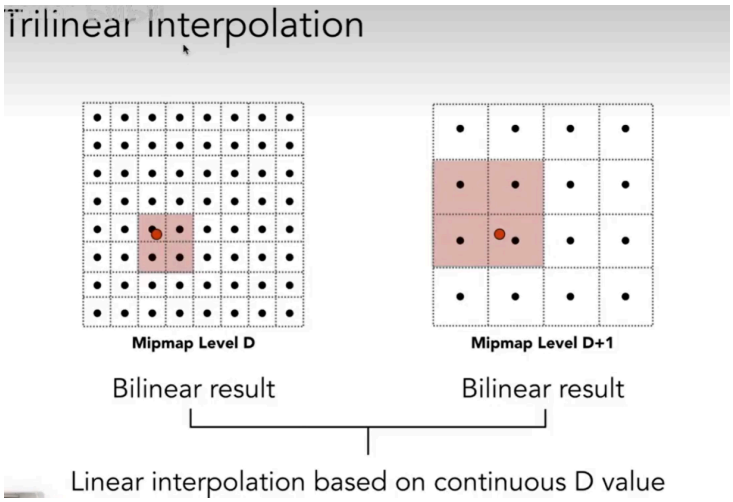
这里的三次插值把周围的四个 texel 的值都考虑到了，一共是两个方向，所以叫双线性插值。

MipMap

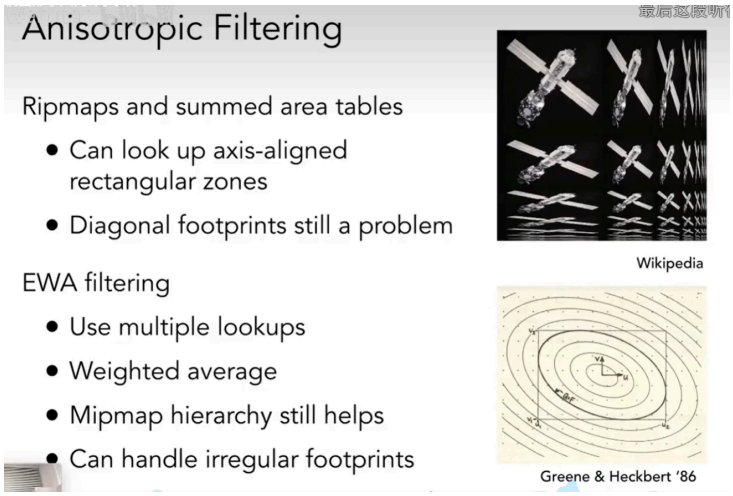
非常快，存在误差，只能做正方形的范围查询，空间换时间，存储空间仅增加 1/3，如下：



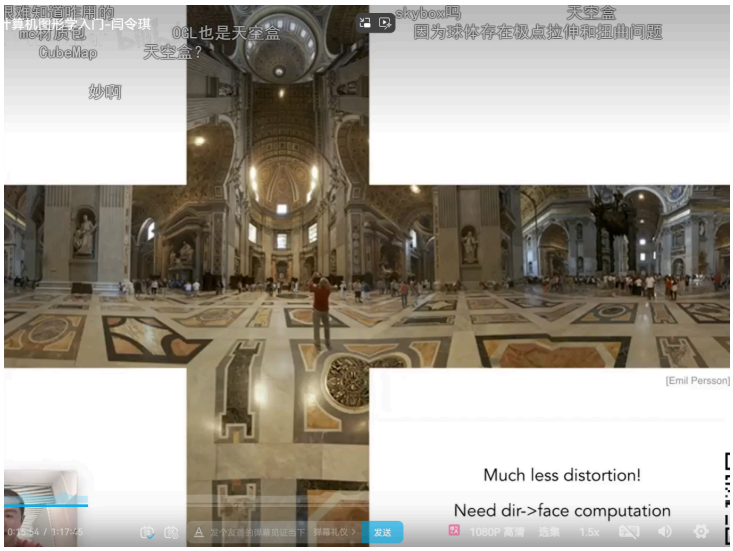
霍霍，mipmap 的两层之间的查询结果也可以做插值，三线性插值！



还有各向异性过滤，



记录环境光的天空盒：



凹凸贴图和法线贴图

定义不同位置的高度，通过临近位置的高度重新计算法线，就是凹凸贴图。

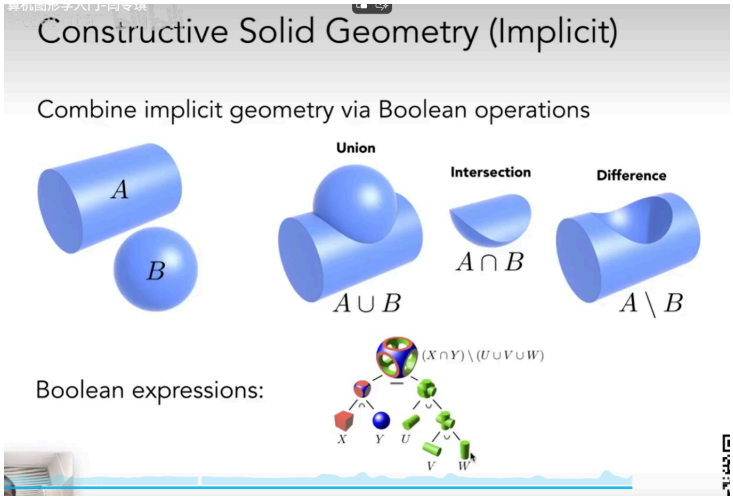
直接存储法线的是法线贴图。

之后的 3d 纹理之类的也就是听个响，直接快进到几何吧。

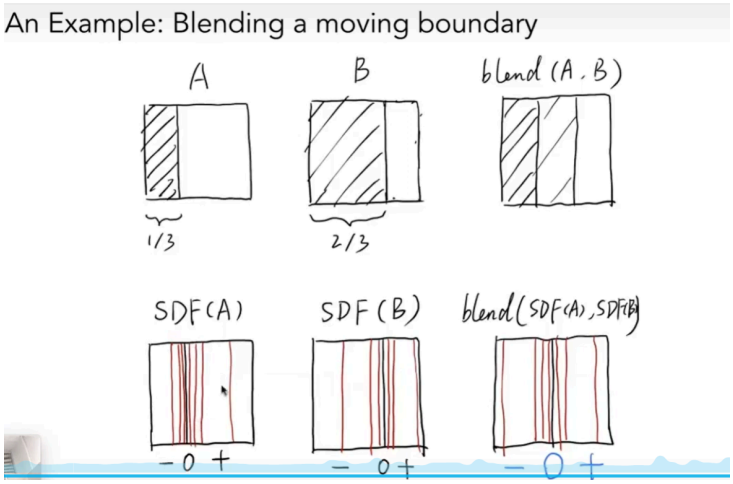
隐式表示就是只给出约束，实际上的点要自己解析出来。

显式表示则难以判断内外。

下面的不错。



下面的图说明 signed distance function(SDF) 的 blend(混合) 可以准确求出中间状态



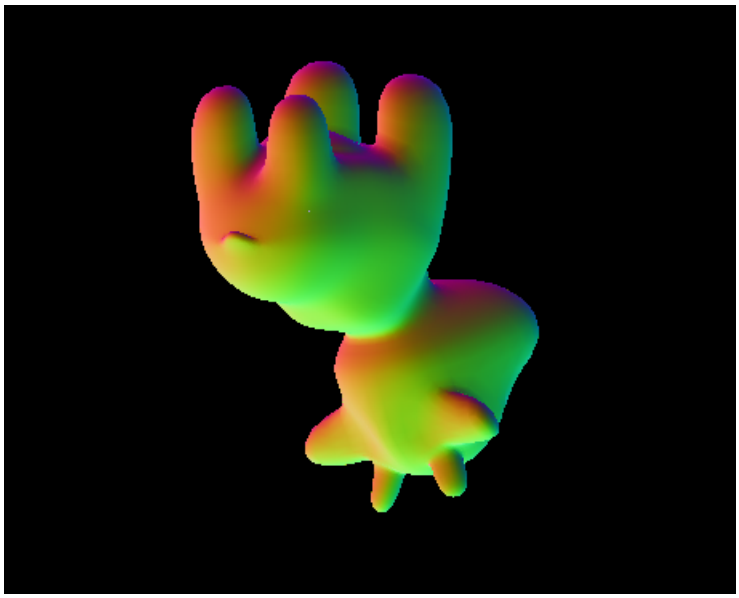
这两节课除了插值的部分感觉都像是通识课。不管了，做作业，接下来是作业 3。

妈的我做破防了，前两个要求写完都运行不出来结果是不能从图形界面直接点运行，因为需要传命令行参数，啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊😊😊😊😊😱😱😱🤗🤗🤗🤓🤓🤓😎😎🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡🤡

首先前两个要求没啥好说的，就是直接用以前的代码算完插值丢到 shader 里去。

```
auto[alpha, beta, gamma] = computeBarycentric2D(x, y, t.v);
float w_reciprocal = 1.0/(alpha / v[0].w() + beta / v[1].w() + gamma / v[2].w());
float z_interpolated = alpha * v[0].z() / v[0].w() + beta * v[1].z() / v[1].w() + gamma * v[2].z() /
z_interpolated *= w_reciprocal;


fragment_shader_payload payload;
//payload.view_pos = interpolated_shadingcoords;
//payload.view_pos =
auto pixel_color = fragment_shader(payload);

if(z_interpolated < depth_buf[idx]) {
 auto interpolated_color = alpha * t.color[0] + beta * t.color[1] + gamma * t.color[2];
 auto interpolated_normal = alpha * t.normal[0] + beta * t.normal[1] + gamma * t.normal[2];
 auto interpolated_texcoords = alpha * t.tex_coords[0] + beta * t.tex_coords[1] + gamma * t.tex_coor
 fragment_shader_payload payload( interpolated_color, interpolated_normal.normalized(), interpolatec

 auto interpolated_shadingcoords = alpha * view_pos[0] + beta * view_pos[1] + gamma * view_pos[2];
 payload.view_pos = interpolated_shadingcoords;
 auto pixel_color = fragment_shader(payload);

    depth_buf[idx] = z_interpolated;
    frame_buf[idx] = pixel_color;
    // 这里其实不能写 .color
}
```

然后 `./rasterizer output.png normal` 运行，嗯哼



浅浅瞅一下命令行参数的接口

```
    if (argc == 3 && std::string(argv[2]) == "texture")
    {
        std::cout << "Rasterizing using the texture shader\n";
        active_shader = texture_fragment_shader;
        texture_path = "spot_texture.png";
        r.set_texture(Texture(obj_path + texture_path));
    }
    else if (argc == 3 && std::string(argv[2]) == "normal")
    {
        std::cout << "Rasterizing using the normal shader\n";
        active_shader = normal_fragment_shader;
    }
    else if (argc == 3 && std::string(argv[2]) == "phong")
    {
        std::cout << "Rasterizing using the phong shader\n";
        active_shader = phong_fragment_shader;
    }
    else if (argc == 3 && std::string(argv[2]) == "bump")
    {
        std::cout << "Rasterizing using the bump shader\n";
        active_shader = bump_fragment_shader;
    }
    else if (argc == 3 && std::string(argv[2]) == "displacement")
    {
        std::cout << "Rasterizing using the bump shader\n";
        active_shader = displacement_fragment_shader;
    }
}
```

然后稍微扒一下涉及的其它接口。

```
//首先 main.cpp 中上面的命令行参数接口下面就是设定我们光栅化器的两个 shader
r.set_vertex_shader(vertex_shader);
r.set_fragment_shader(active_shader);
//但我们查一下这个顶点着色器的成分就可以发现这玩意在程序里根本没有用到，
//不仅 shader 本身只有一行，payload 也是只有一行
// in main.cpp
Eigen::Vector3f vertex_shader(const vertex_shader_payload& payload)
{
    return payload.position;
}
// in shader.hpp
struct vertex_shader_payload
{
    Eigen::Vector3f position;
};
```

没啥好扒的，直接继续做吧，首先是 **phong's model**。

这个库其实很方便的，比如算个距离直接

```
float __dis(Eigen::Vector3f a, Eigen::Vector3f b) {
    return (a-b).norm();
}
```

然后是代码和效果

```
for (auto& light : lights)
    {
        // TODO: For each light source in the code, calculate what the *ambient*,
*diffuse*, and *specular*
        // components are. Then, accumulate that result on the *result_color* object.
        auto r = __dis(light.position, point);
        auto l = (light.position - point).normalized();
        auto diffuse = kd * (light.intensity.norm() / (r * r)) * std::max(0.0f,
normal.dot(l));
        auto ambient = ka * amb_light_intensity.norm();

        auto h = (l + (eye_pos - point).normalized()) / 2;
        auto specular = ks * (light.intensity.norm() / (r * r)) *
std::pow(std::max(0.0f, normal.dot(h)), p);
        result_color = result_color + diffuse + ambient + specular;
    }
// std::max 挺难崩的，传 0.0 还不行，估计给我自动转成 double 了，得是 0.0f
```

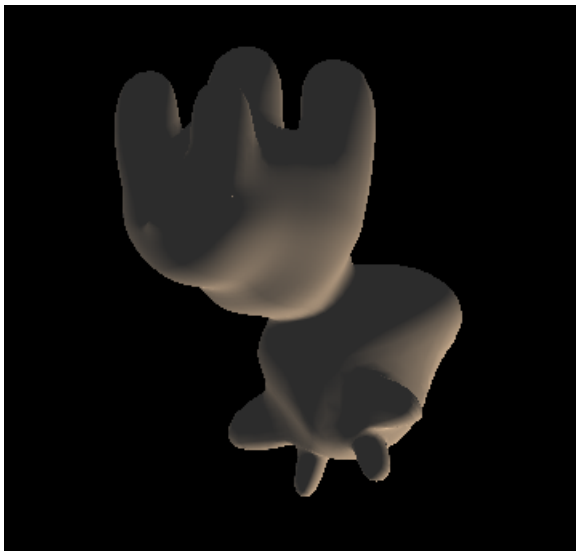


图 1　难绷牛屁股

接下来那个把纹理加进去的其实就多了一行代码，因为框架帮你把 kd 换了。

```
if (payload.texture)
{
    // TODO: Get the texture value at the texture coordinates of the current fragment
    return_color = payload.texture->getColor(payload.tex_coords.x(),
payload.tex_coords.y());
}
```

接下来是凹凸贴图。

```
Eigen::Vector3f n = normal;
auto x = n.x(), y = n.y(), z = n.z();

Eigen::Vector3f t{ x*y/std::sqrt(x*x + z*z), std::sqrt(x*x + z*z), z*y/std::sqrt(x*x + z*z) };
Eigen::Vector3f b = n.cross(t);
Eigen::Matrix3f TBN;
TBN << t, b, n;
auto dU = kh * kn * (_h(u+1.0f/w,v,payload)-_h(u,v,payload));
auto dV = kh * kn * (_h(u,v+1.0f/h,payload)-_h(u,v,payload));
Eigen::Vector3f ln{-dU, -dV, 1.0f};

normal = (TBN * ln).normalized();

Eigen::Vector3f result_color = {0, 0, 0};
result_color = normal;

return result_color * 255.f;
```

哼哧哼哧写了一阵子却发现少了两个变量，傻眼了。但考虑到这里是求偏导，实际上应该把 w 和 h 设成大一点的 float，但实测设成 1.2 都不成，所以还是直接换成 1.0 了。(顺便一提，调成 0.99 效果会有变化)

```
float _h(float u, float v, const fragment_shader_payload& payload) {
    Eigen::Vector3f return_color = {0, 0, 0};
    return_color = payload.texture->getColor(u, v);
    return return_color.norm();
}
// in bump_fragment_shader
float kh = 0.2, kn = 0.1;
auto u = payload.tex_coords.x(), v = payload.tex_coords.y();
Eigen::Vector3f n = normal;
auto x = n.x(), y = n.y(), z = n.z();

Eigen::Vector3f t{ x*y/std::sqrt(x*x + z*z), std::sqrt(x*x + z*z), z*y/std::sqrt(x*x
+ z*z) };
Eigen::Vector3f b = n.cross(t);
Eigen::Matrix3f TBN;
TBN << t, b, n;
float w = 1.0f, h = 1.0f;
auto dU = kh * kn * (_h(u+1.0f/w,v,payload)-_h(u,v,payload));
auto dV = kh * kn * (_h(u,v+1.0f/h,payload)-_h(u,v,payload));
Eigen::Vector3f ln{-dU, -dV, 1.0f};

normal = (TBN * ln).normalized();

Eigen::Vector3f result_color = {0, 0, 0};
//result_color = {0.33f, 0.33f, 0.33f};
result_color = normal;

//std::cout<<result_color.x()<<' ';

return result_color * 255.f;
```

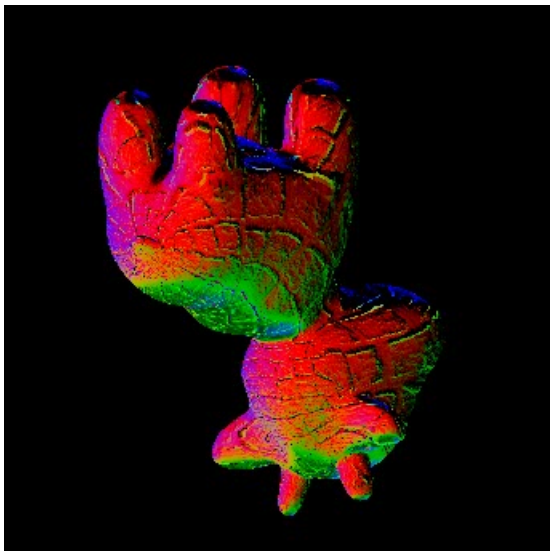


图 2　w=h=1.0，每次渲染出来可能稍微有点不一样

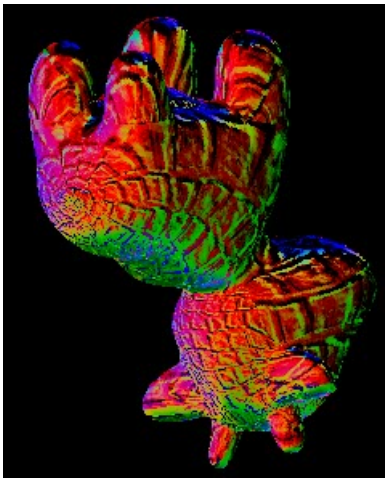


图 3　w=h=0.99

然后 displacement_shader 其实就是用 bump 的法线做 phong_shading，然后这里如果你有兴趣也可以加上 texture，虽然看着一般。