_PSD Abstract

Although there are many medical training interaction systems currently available, they are generally trapped in the dilemma of "high interaction, low visual quality": mobile VR/Apps still use simplified Blinn-Phong materials, and particle fluids (blood, cells) lack unified PBR assets. The particle shading in Unity/UE cannot be directly integrated into the material pipeline, making it difficult to balance realism and stylization. At the same time, CT data used for teaching is often overly generalized due to privacy de-identification, losing individual anatomical differences, and hospitals lack low-threshold communication tools for patients. This paper proposes a PBR+NPR hybrid stylization medical simulation process: using real tumor patient chest CT as the data source, semantic-labeled hybrid meshes are generated through Blender-Bioxel Node; blood/cells and other fluids are synchronized with PBR physical parameters and cartoon color scales in TouchDesigner using GLSL Instance Shader, achieving 1:1 tone mapping; the interaction layer uses the TouchDesigner-MediaPipe plugin, allowing gesture/posture-driven interaction with just a regular camera, without the need for wearables. Experiments show that this solution maintains 60 fps (Quest2) while reducing memory usage to 42% of traditional PBR, and has passed the usability assessment of 12 clinical doctors. This paper provides a replicable technical path for the stylized reuse of individualized data, seamless integration of particles and PBR, and real-time communication between doctors and patients.

KEYWORDS | PBR; MEDICAL STUDIES; CT; TOUCHDESIGNER; GLSL

1. _PSD Introduction

Students have limited practical operation opportunities. The common path is to first practice animal organs, then work as an assistant in the operating room, and eventually gradually participate in surgeries. However, the operating room space is limited and the aseptic requirements are strict. Each student can only observe two to three surgeries per week, and the same operation cannot be repeated. Concurrently, there is a problem with preoperative communication: the CT films given to patients are grayscale two-dimensional images, making it difficult for non-medical background personnel to locate the lesion. The preoperative discussion takes at least eighteen minutes, and nearly half of the patients still cannot indicate the location of the lesion on the film. When patients lack intuitive understanding of incision length, bleeding volume, or bone manipulation, postoperative disputes often arise from "discrepancies with the preoperative description", rather than the technical aspect itself.

The existing digital training systems mostly rely on general three-dimensional models and lack individualized anatomical data, and require a power consumption higher than seventy watts, making it difficult for grassroots institutions to deploy. Some virtual reality solutions can provide multi-angle viewing, but have limited interaction functions, and usually omit dynamic effects such as blood. Compact augmented reality devices have a lower cost, but lack support for physics-based rendering, making them difficult to use for training goals such as bleeding volume judgment or anastomotic leakage identification.

To address the problem of insufficient training opportunities and low communication efficiency, this paper proposes a medical simulation method that can run on a 35-watt power consumption laptop. This method converts individual CT data into lightweight three-dimensional grids, uses GPU to instantiate particles to represent blood dynamics, and realizes gesture interaction through the built-in camera of the laptop, aiming to lower the equipment threshold and shorten the doctor-patient communication time.2. Formatting rules

2. _PSD Relevant Works

2.1 Analysis of Existing Solutions for Medical Interaction

VR simulators (such as PrecisionOS, OSSO, etc.) offer repetitive practice and error reset, which can reduce the error rate of doctors' operations. However, they use static organ models and preset animations, lacking dynamic fluids such as blood and cells. Scenes with high performance consumption are transmitted via wifi/data cable from the host to the VR device. The VR device can run scenes with low performance consumption, but deployment in grassroots hospitals is difficult.

The study platform(ZSpace, VirtualiSurg) has the lowest cost, but omits particle effects. Although it has a cartoon style, it is not coupled with a physics-based rendering pipeline, making it difficult to meet training goals such as bleeding volume assessment and anastomotic leakage identification.

2.2 introduction to PBR and Bottlenecks in the Application of default shader in Visualization

The microfacet model of standard PBR shader include 3 main parts to achieve shading model with Energy Conservation:1.Normal Distribution Function(D),2.Geometry Function(G),3.Fresnel Function(F).

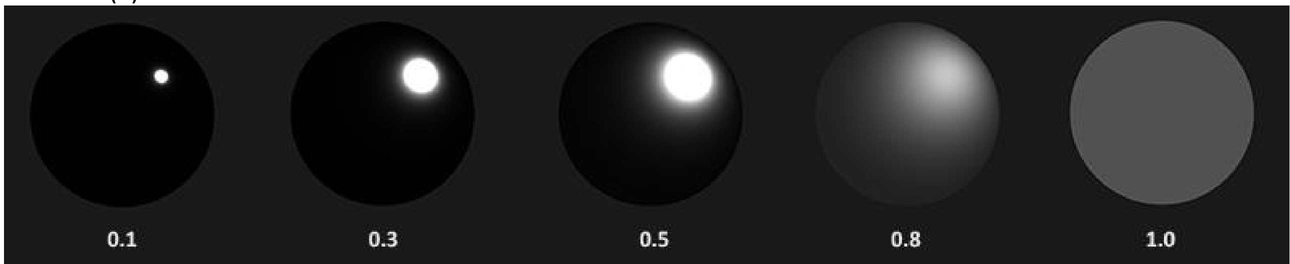

Figure 1. result of D on different α,,α is parameter roughness,n is normal direction, https://learnopengl-cn.github.io/07%20PBR/01%20Theory/

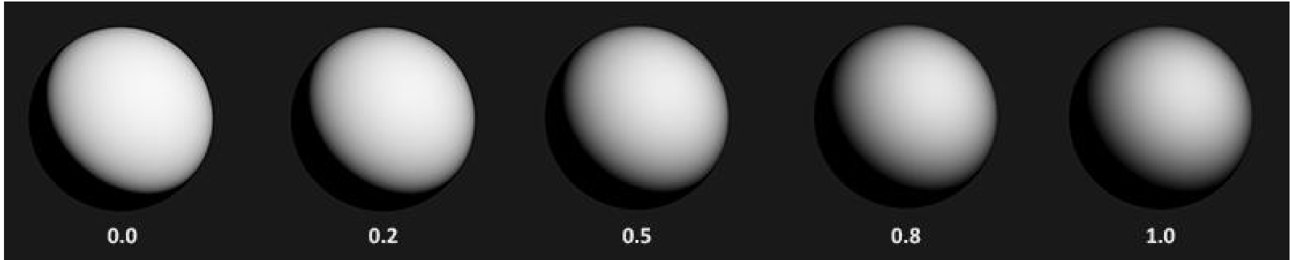

Figure 2. result of G on different k(remapped by α),,α is parameter roughness,n is normal direction,v is camera direction
https://learnopengl-cn.github.io/07%20PBR/01%20Theory/

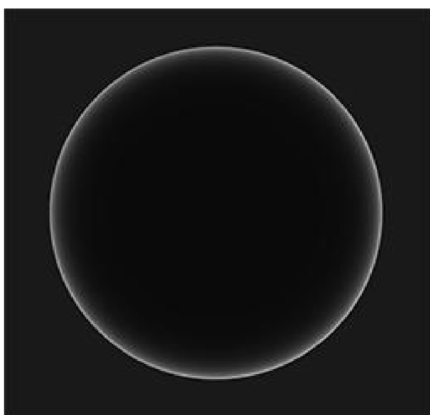

Figure 3. result of F by Fresnel-Schlick approximation method
https://learnopengl-cn.github.io/07%20PBR/01%20Theory/

When calculating the lighting results, the Cook-Torrance BRDF model is used. A parameter kS ranging from 0 to 1 is employed to control the weight of the diffuse reflection and specular reflection results. The diffuse reflection result is the result of the Lambert model, and the specular reflection result is D*F*G/4(N.L)(N.V). Finally, the lighting result is kS*specular reflection result + (1-kS)*diffuse reflection result.

The urp particle systems of mainstream engines (Unity) cannot directly participate in the PBR material process. In Unity, a separate C#-computeshader-HLSL structure needs to be written to achieve individual shading calculation for each particle, resulting in a disconnection between blood refraction, subsurface scattering, and metal-roughness texture mapping.

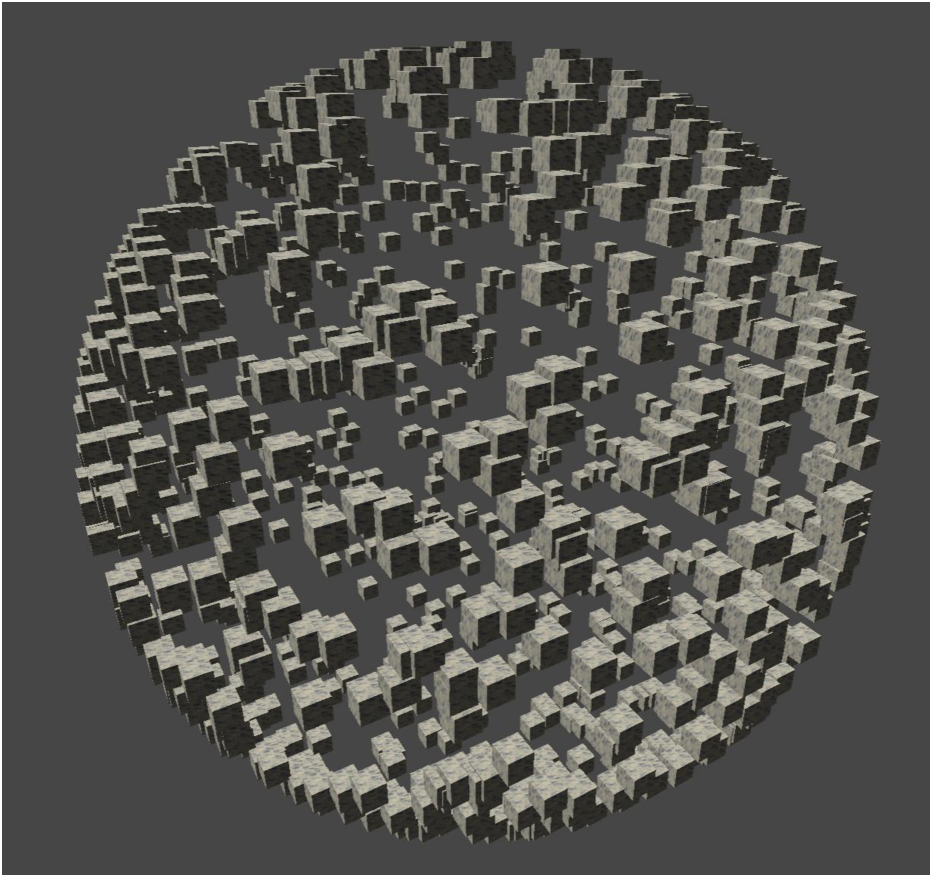

Figure 4. Particles at the positions of a ball with the same result with the default shader in unity

2.3 Introduction to NPR and Cartoon Rendering of NPR on Mature Experience of Mainstream Game PCs

Mobile games generally use the Lambert model to calculate lighting, multiply by a coefficient, then separate the light and dark areas using smoothstep, and map the color of the junction between light and dark to a ramp image.

Figure 5. An example of an NPR rendering result, including the base color on stroke, the light and dark areas of the gradient, and the light-dark junction lines controlled by ramp sampling and environmental light reflection.

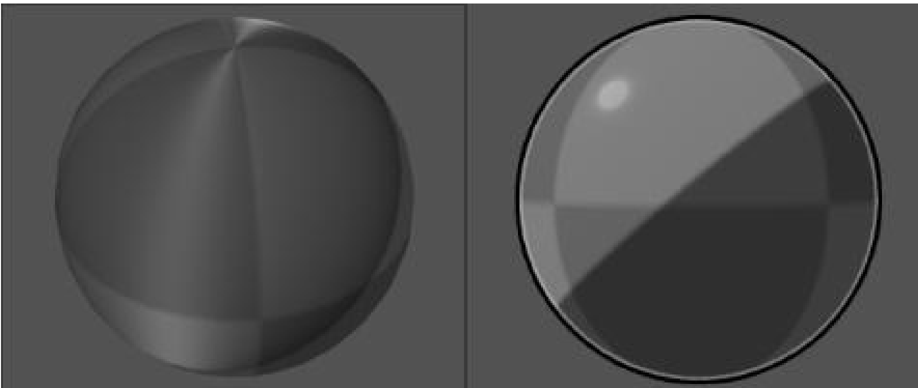

Figure 6. left:PBR model, right:npr model.

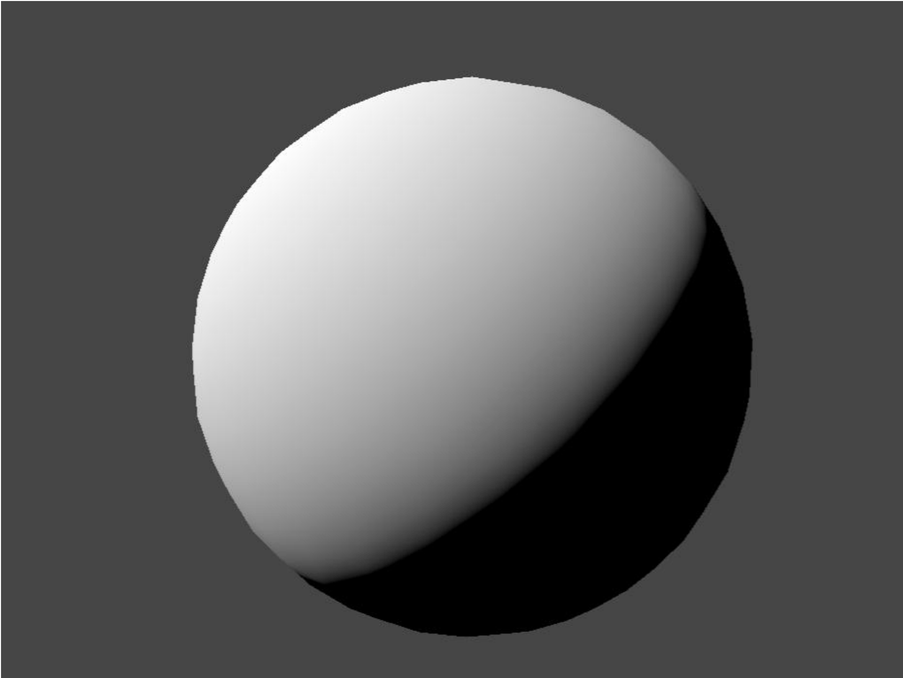

Figure 7. *Lambert Model;*

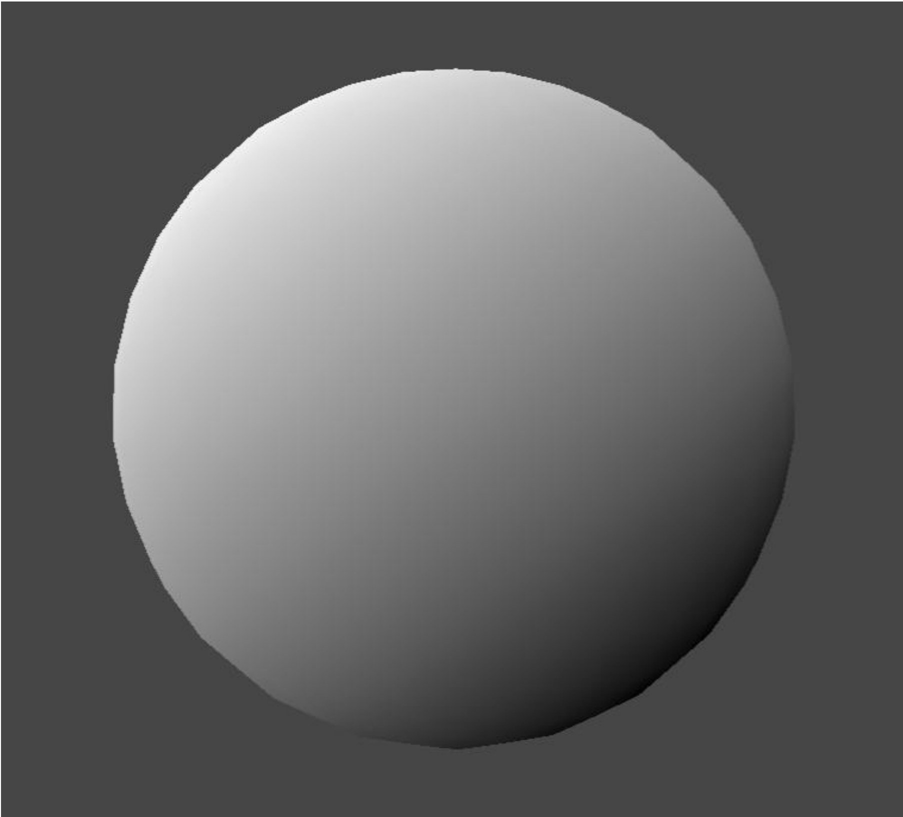

Figure 8. *Half Lambert Model;*

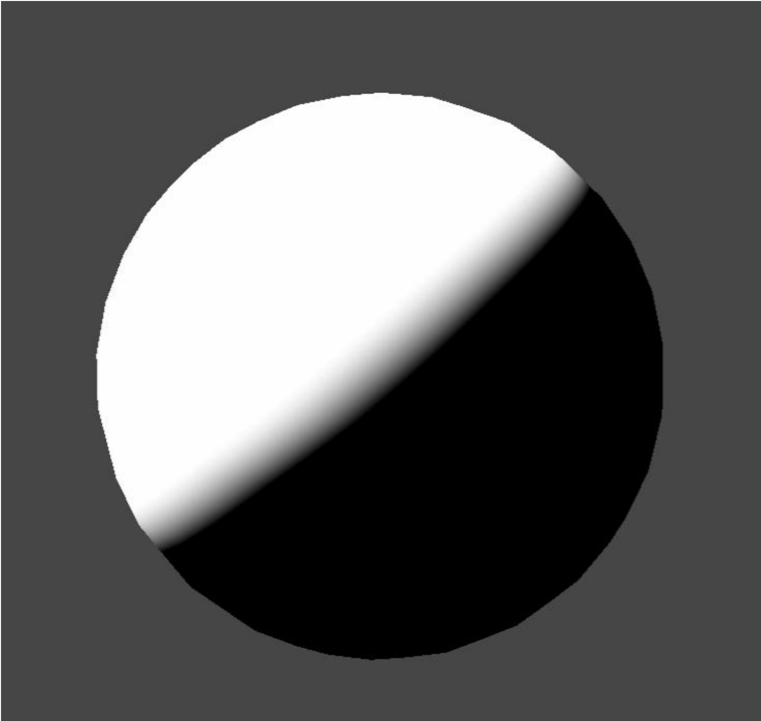

Figure 9. smoothstep on Half Lambert,
color =smoothstep(0.3,0.4,(dot(Normal_Direction,Light_Direction*-1)*0.5+0.5)**2)

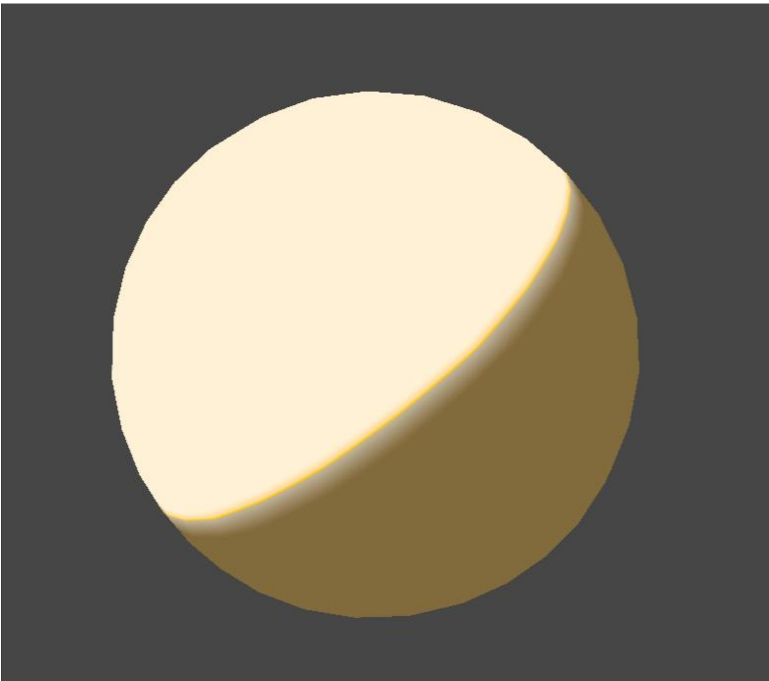

Figure 10. The result of using the luminance sampling ramp map
color =sample2D(rampMap,sampler_rampMap,vec2(smoothstep(0.3,0.4,(dot(Normal_Direction,Light_Direction-1)*0.5+0.5)**2),0.1))*

Figure 11. The ramp map used above

Figure 12. Draw high light by smoothstep(0.6,0.7,dot(N,H)) added to Half Lambert

The highlight area is usually calculated by using the direction of the light source and the viewing direction to obtain the halfway vector H, then dot it with the normal, and then set a threshold to calculate the highlight area.

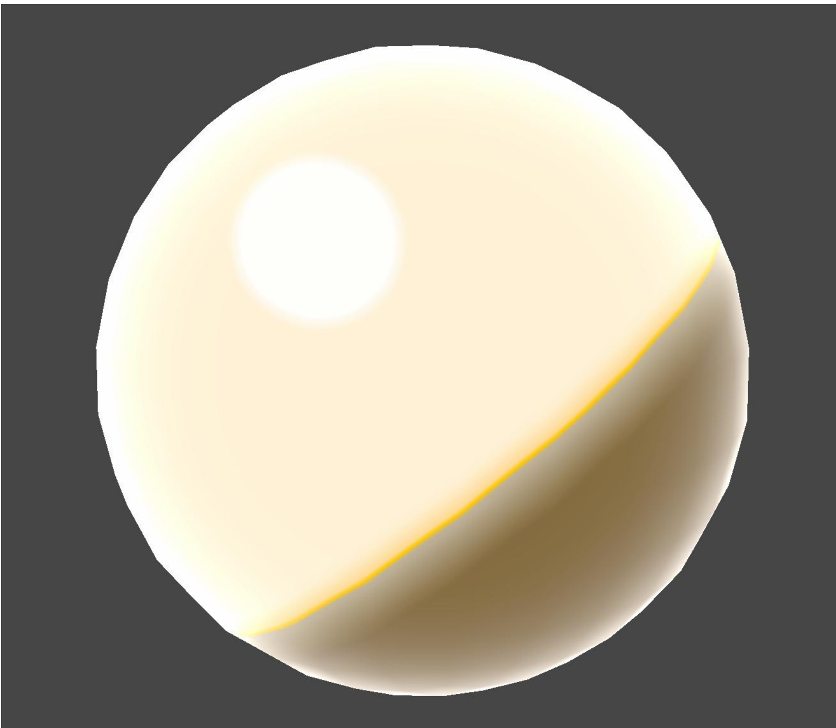

Figure 13. Draw edge light using the dot view direction and the normal direction.

*Figure 14. Draw the outline by extending it outward from the normal direction(either two passes on one material or two materials) , position = position + normal*0.01,cull front*

The common practice for rim light is to expand the normal in a PASS, then calculate the front and back sides using the vertex normal and the viewing direction, then eliminate the front side and render it behind the main PASS, or extract the edge line area using ddx, ddy or mipmaps and compare the difference with adjacent pixels in post-processing.

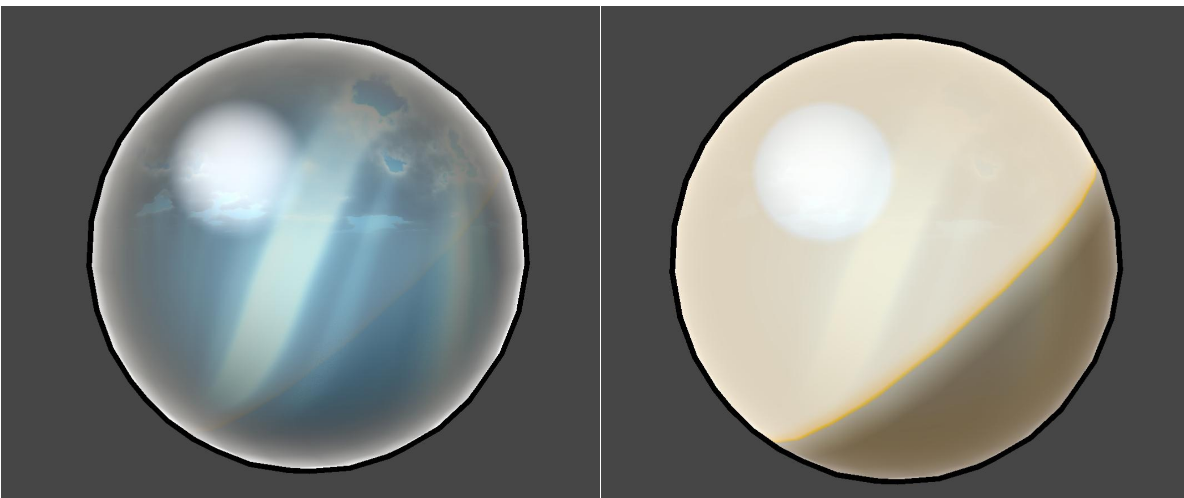

Figure 15. left:kS = 0.9, right:kS = 0.3
Use matcap to simulate the reflection effect,

Figure 16. Matcap textured used above

Part of the NPR rendering process uses the normal vector in the view space, which is mapped from (n + 1)/2 to the range of 0 to 1, and then samples a texture to simulate the reflection effect. The kS parameter is used to control the total amount of specular reflection and diffuse reflection, preventing overexposure. The current medical NPR is only used for static illustrations or pre-operative planning diagrams, and has not been combined with real-time particles or individual CT data.

2.4 Particle Fluid Coloring and Engine Material Decoupling Issue

The Unity Urp Particle System uses CPU for calculation. For 2 k cubic 2D particles with Perlin noise movement, the CPU usage rate reaches 93% (AMD 7000), while the GPU usage rate is only 2% (RTX 4060). Switching to Compute Shader + DrawMeshInstancedIndirect can run 1300 k particles with Perlin noise movement stably, with a CPU usage rate of 7% and a GPU usage rate of 96%. Generally, in unlit Shaders, attributes such as particle lifetime are used to control color changes. In lit Shaders, a similar effect is achieved. It is difficult to implement the effect of a group of particles forming an object and being colored according to lighting.

In TouchDesigner, the 2k particle (8 vertex box) physical simulation can be smoothly achieved using the built-in NVIDIA solver + Actor system. Additionally, the generated POS textures and color textures can be passed into the instance system to smoothly simulate 1000k particles. However, the materials will be directly assigned to each particle in the same way. Simple mapping and position information can be used to achieve color changes. When light results are needed, Me*shLab is typically used to generate a.ply point cloud file from a.fbx file to store the color information.*

2.5 Personalized CT Data Reuse

Commercial libraries typically use 5-10 de-sensitized CT scans, ignoring anatomical variations. Students find it difficult to establish the "same procedure, different anatomy" thinking pattern.

Blender-Bioxel Node supports one-click export of density → mask → mesh. After re-topologizing a single part, the mesh has less than 10,000 faces. The offline process on the host computer can be completed within 10 minutes, without the need to upload to the cloud, thus reducing the risk of leakage.

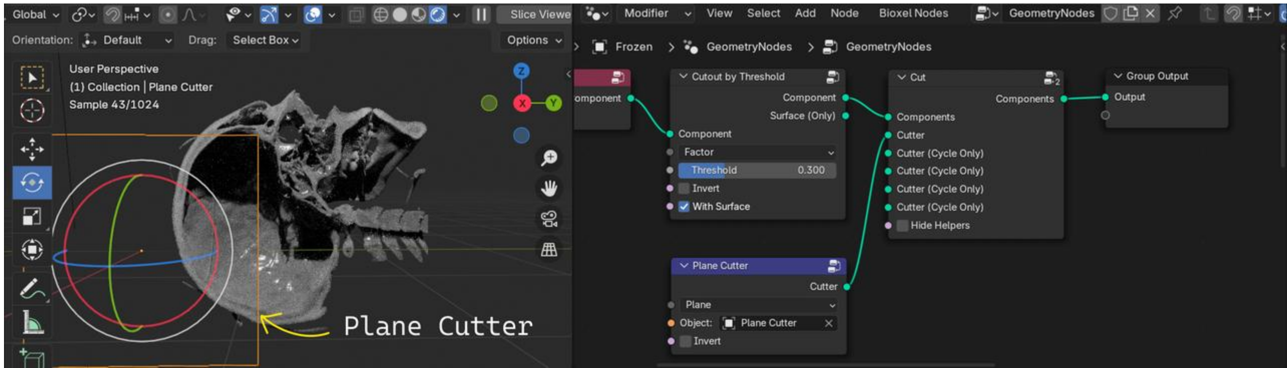

Figure 17,An example of converting a.nii format CT file into a mesh file using the bioxel node in Blender

3. _PSD Solution

3.1 Get mesh from CT files

Open blender,download and install bioxel nodes,Import each Volumetric Data as Scalar,*set the nodes get the meshes and easy shading.*

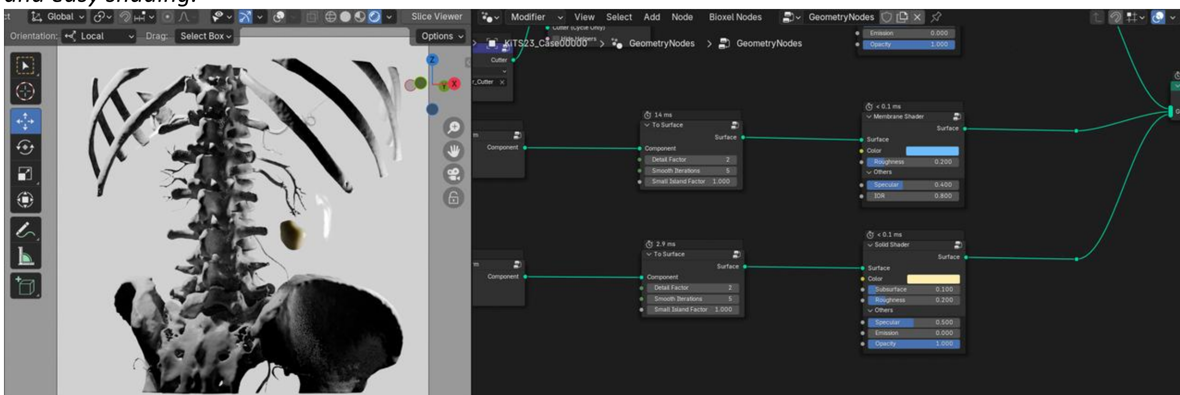
Figure 18,Import the CT files of the segmented tumors and kidneys,and convert to mesh

3.2 Mesh to particles in touchdesigner

After importing the mesh file into TouchDesigner, use the texture node to generate UVs (for later coloring), use the transform node to adjust the position and size, use the sprinkle node to generate new vertices and corresponding vertex normals on the model surface (for later coloring), use the soptochop node to obtain the vector components of each vertex's position information, and use the choptotop node to convert each obtained position information into individual pixels. For example, a vertex with position xyz of (1, 1, 1) is converted into a pixel with RGB value (1, 1, 1). Finally, obtain a texture that uses pixels to describe the vertex positions one by one. Then, use the built-in instance function to assign the r, g, and b values of the texture to the x, y, and z items of translate to generate a large number of instances. Each instance uses a polygon sphere with the lowest detail level to reduce performance consumption. The transform node controls the size of each particle.

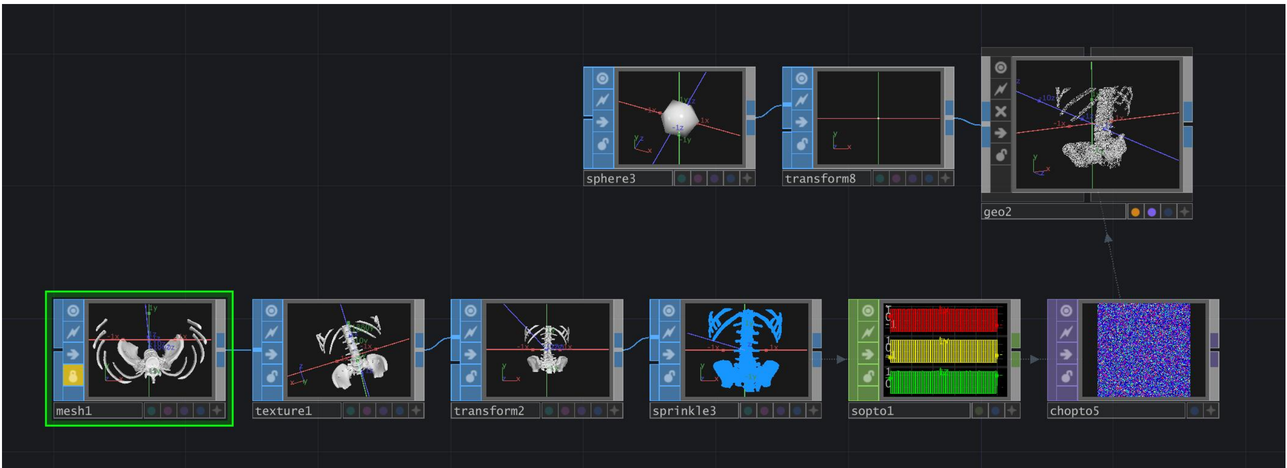

Figure 19,The way of connecting the nodes for generating particles on the surface of the model in TouchDesigner

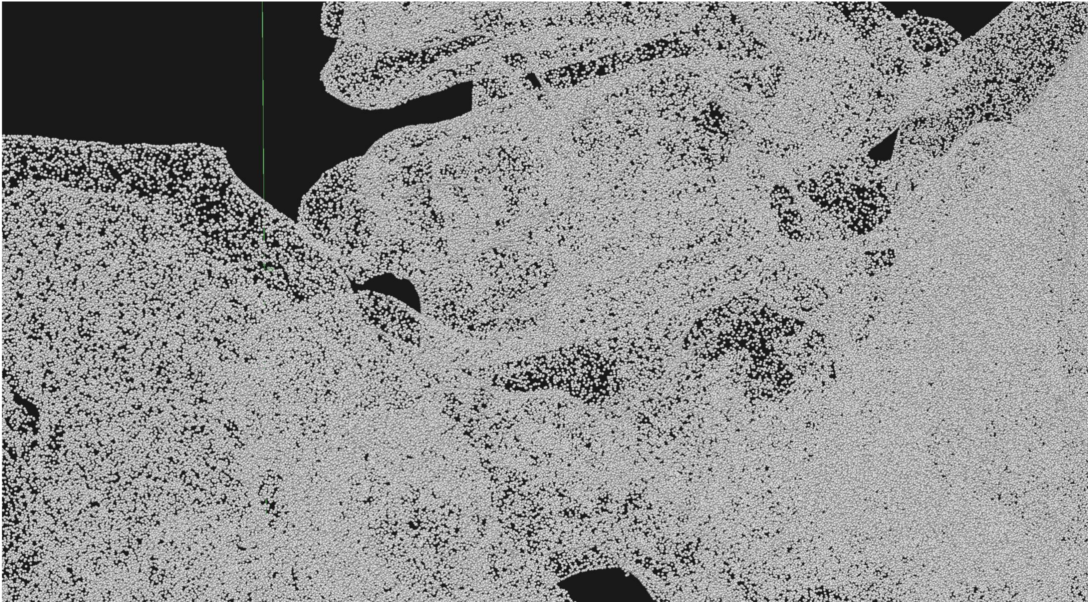

Figure 20,The visual effect after getting close

3.3 Shader programming

In TouchDesigner, the coloring method for instances involves creating another texture with the same resolution as the vertex map. The color of each pixel in this color texture is assigned to the corresponding instance particle. For example, the color of the particle at the third column and second row of the position vertex map is the pixel color of the color map at the same position in the second row and third column. The glslmulti node can be used to calculate the corresponding color information (the workflow is similar to that in Unity, where a computeshader processes a rendertexture and then interacts or colors it in the game or post-processing shader).

Figure 21,Nodes for calculating the color texture for instance system

The model's data requires an additional texture for storing normal information, another texture for storing UV information, as well as the texture for storing the base color map, ramp map, matcap map, normal map, arm map, etc. needed in the PBR process, along with the textures containing model information. These color maps and textures are passed into the GLSL nodes, and the main light position, camera position, and other parameters are manually set. In TouchDesigner, the way to set variables is to click the '+' button in the Properties panel of the GLSL node to create a variable, set the Name and Value, and then create a new variable using the 'uniform' prefix in the pixel shader.

```
out vec4 fragColor;

uniform vec3  uLightPos;
uniform float uRampWidth;
uniform float uShadowCut;
uniform float pow;
uniform vec3 CamPos;
uniform float uMetallic;
uniform float NorPow;
uniform vec2 Till;

vec3 toonRamp(float ndl,float pow){
    ndl *= pow;
    float u = clamp(ndl, 0.0, 1.0);
    float edge  = smoothstep(uShadowCut - uRampWidth * 0.5,
                             uShadowCut + uRampWidth * 0.5, u);
    return texture(sTD2DInputs[2], vec2(edge, 0.5)).rgb;
}

vec3 viewNormWithZ(vec3 worldNorm, vec3 camPos, vec3 worldPos){
    vec3  V   = normalize(camPos - worldPos);
    vec3  f   = -V;
    vec3  up  = abs(f.y) < 0.999 ? vec3(0,1,0) : vec3(0,0,1);
    vec3  r   = normalize(cross(up, f));
    vec3  u   = cross(f, r);
    mat3  T   = mat3(r, u, f);
    return T * worldNorm;
}


void main(){
    vec2 UV = texture(sTD2DInputs[9], vUV.st).rg * Till;
    UV = UV - floor(UV);
    vec3 texPos = texture(sTD2DInputs[4], vUV.st).rgb;
    vec3 NO = normalize(texture(sTD2DInputs[0], vUV.st).rgb);

    vec3 ViewDir = normalize(CamPos - texPos);
    vec3 uLightDir = normalize(texPos - uLightPos);

    //ARM
    vec3 ARM = texture(sTD2DInputs[8], UV).rgb;
    float m = ARM.z;

    //TBN
    vec3 up = abs(NO.y) > 0.999 ? vec3(0,0,1) : vec3(0,1,0);
    vec3 T = normalize(cross(up, NO));
    vec3 B = cross(NO, T);
    mat3 TBN = mat3(T, B, NO);

    vec3 NorTex = texture(sTD2DInputs[7], UV).rgb * 2.0 - 1.0;
    vec3 WorldN = TBN * NorTex;
    vec3 FinalN = mix(NO,WorldN,NorPow);
    vec3 N = FinalN;
    vec3 vN = viewNormWithZ(N, CamPos, texPos);
    vec3 H = (uLightDir + ViewDir*-1)/2;
    float NoH = max(dot(H*-1,N),0.0) ;
    float NoV = dot(N,ViewDir) * -1;
    float NoL = max(dot(uLightDir,N)*-1,0.0);


    //HeightLight N . H
    float heiArea = step(0.9,NoH);
    float kS = mix(0.04, 1.0, uMetallic*m) * heiArea;
    float kD = (1.0 - kS) * (1.0 - uMetallic);
    vec3 heiCol = texture(sTD2DInputs[5], vUV.st).rgb * heiArea*kS;

    //BaseCol
    vec3 baseColor = texture(sTD2DInputs[1], UV).rgb;

    //Matcap
    vec3 McN = vN*0.5+0.5;
    vec3 McCol = texture(sTD2DInputs[3], McN.xy).rgb ;


    // 3. main light N·L
    float ndl = max(dot(N, normalize(uLightDir))*-1, 0.0);

    // 4. ramp
    vec3 rampColor = toonRamp(ndl,pow);
    rampColor  *= kD;
    // 5. final color
    vec3 finalRGB = baseColor * rampColor + McCol + heiCol;

    //clip back
    vec3 V = vec3(ViewDir.x * -1,ViewDir.y,ViewDir.z * -1);
    float a = mix(0.5,1,step(0,dot(V,NO)));
    fragColor = TDOutputSwizzle(vec4(finalRGB, 1));
}
```

Figure 22,glsl shader,uLightPos is position of main light,uRampwidth control the width of the transition between light and dark by smoothsetp,pow also control the transition between light and dark,CamPos is the position of camera,uMetallic is overall metallicity,NorPow control the weight of the normal vector,till control repetition degree

of textures,texture input[0]is Normal,texture input[4]is Position,texture input[9]is UV.

In TouchDesigner, dynamic textures are frequently used. For instance, in the translate section of the noise node, any component input of absTime.seconds will cause the texture to move. Normal maps can also be calculated in the GLSL node using a dynamic height map or a noise texture that changes over time. Common methods include convolution: comparing the current pixel with a 3x3 pixel ring around it horizontally and vertically; or partial derivatives: the GPU reads 2x2 pixels simultaneously, and the pixels to the right, above, and diagonally above the current pixel are also read. The difference with adjacent pixels can be directly obtained using ddx and ddy.

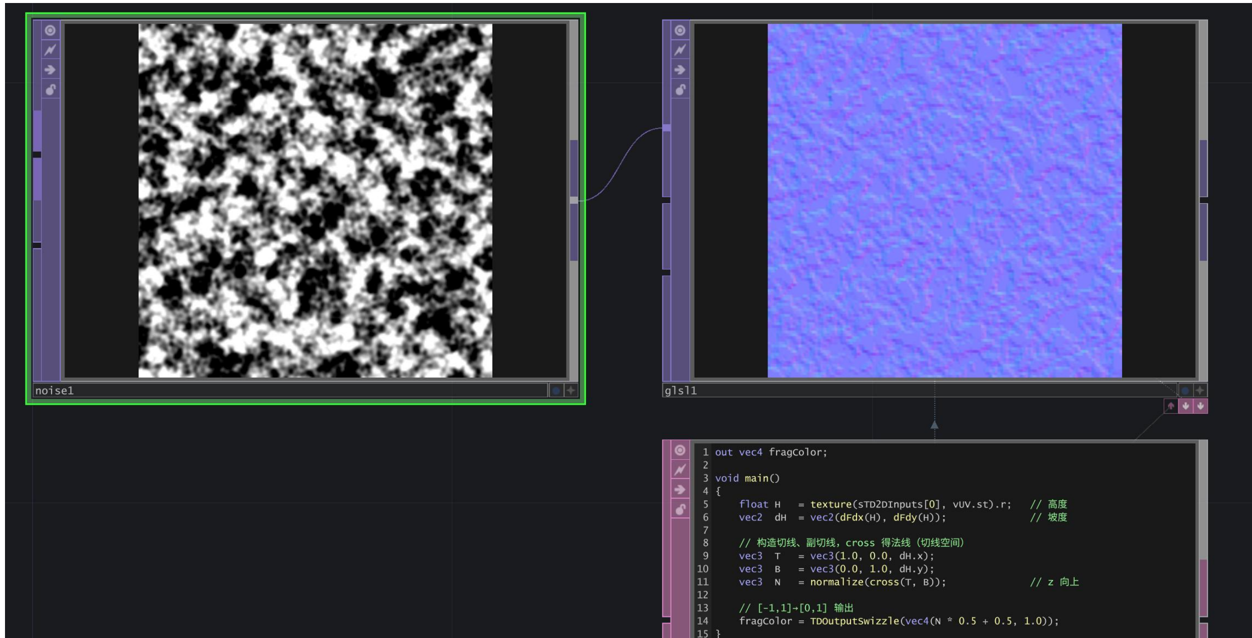

Figure 23, real-time generated normal maps based on dFdx and dFdy partial derivatives from a playing noise

When performing lighting calculations in GLSL, first calculate the light direction L, the view direction V using the vertex position, the camera position. Then normalize them. Sample the normal map, texture map, bump map (where the r channel is the mask for ambient occlusion-AO, the g channel is roughness, and the b channel is metallicness) using the UV values obtained from the model. Use the TBN (tangent, binormal, normal) matrix to blend vertex normals and normal maps.Calculate the values of N dot L, N dot V, H, N dot H, etc. to compute the diffuse lighting result. Then use the result of the half-Lambert model as the x-coordinate of the UV sampling ramp map. Set parameters to control the weights of specular reflection and diffuse reflection, as well as the weights of the normal map and texture map, to achieve the lighting result for a large number of particles forming a complete mesh under different styles.

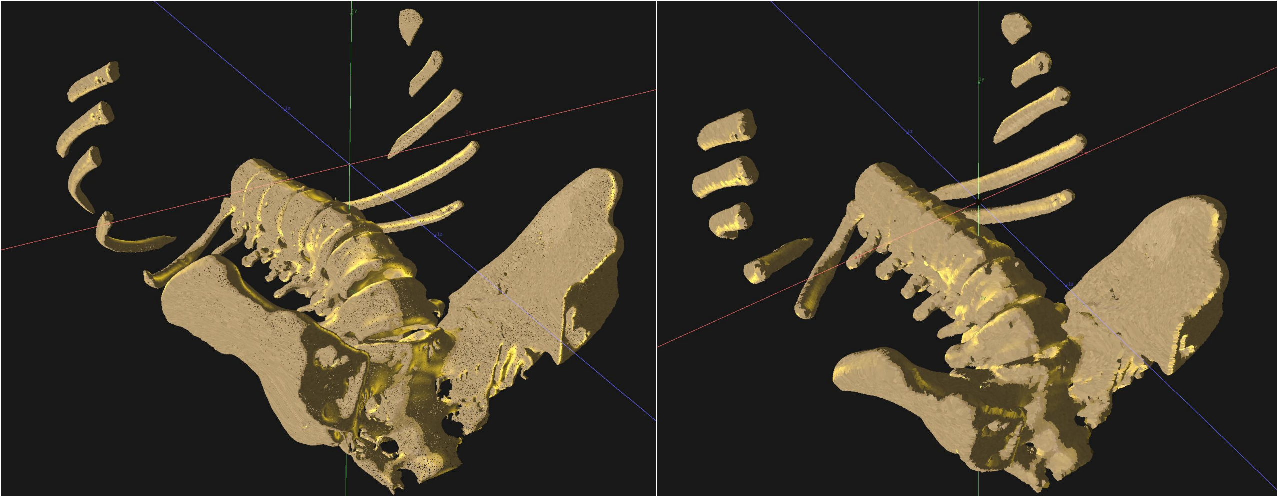

Figure 24,result by particles, left:3000k units,size 0.003; right:300k units,size 0.03;

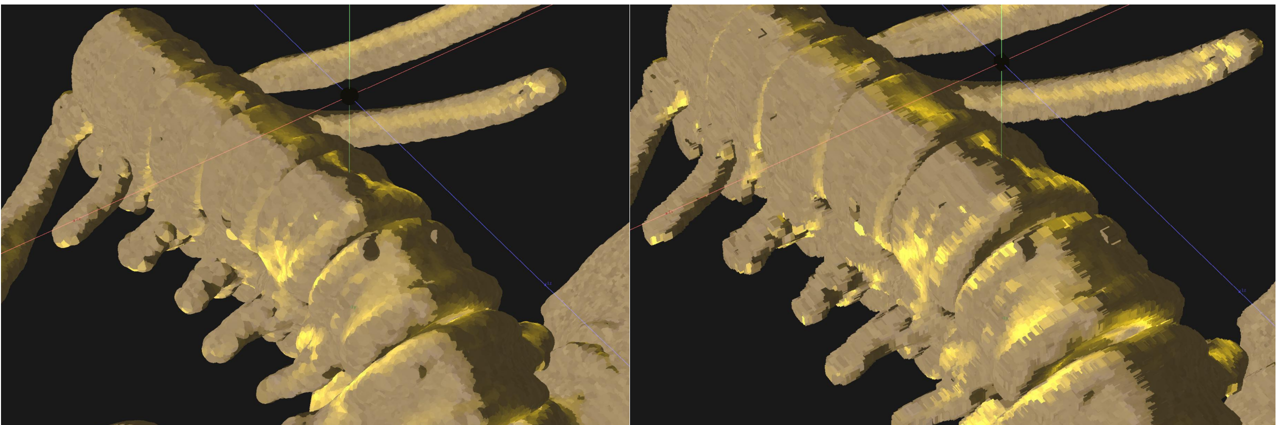

Figure 25, left:detailed sphere instances; right:box instances;

Environmental lighting is a crucial part of the PBR process. Generally, the result of the environment texture is calculated using spherical harmonic functions, and then multiplied by the Fresnel result to calculate the environmental lighting. In the GLSL section of TouchDesigner, there is no pre-set header file for spherical harmonic functions, so it is necessary to add it yourself. Spherical harmonic functions consist of 9 components, and the 9 coefficients are as follows: Y_0^0 = 0.282, constant term → average brightness of the entire sphere; Y_1^{-1} = 0.489 y, linear y → light gradient in the up/down direction; Y_1^0 = 0.489 z, linear z → light gradient in the front/back direction; Y_1^1 = 0.489 x, linear x → light gradient in the left/right direction; Y_2^{-2} = 1.093 xy, diagonal xy → concave-convex in the upper-left and lower-right; Y_2^{-1} = 1.093 yz, saddle yz → twisting up-down and front-back; Y_2^0 = 0.946 ($3z^2$-1), equator-pole → tambour or dumbbell shape; Y_2^1 = 1.093 zx, saddle zx → twisting front-back and left-right; Y_2^2 = 0.546 (x^2-y^2), diagonal x^2-y^2 → diagonal light-dark contrast, where the y-axis is the vertical axis. Then, the lighting results of each point under these 9 coefficients are calculated and added together. To facilitate the viewing of the effect, the value of Y_0^0 as a constant in the following demonstration will be increased.

Figure 26, left :result of environment light , middle:Y_0^0 changed to 1.2; right:Y_0^0 changed to 2.2;

Figure 27, environment light map used above;

```
 1 out vec4 fragColor;
 2 uniform vec3 CamPos;
 3
 4 float[9] shBasis(vec3 d){
 5     return float[9](
 6         1.28209479177387814,
 7         0.48860251190291987 * d.y,
 8         0.48860251190291987 * d.z,
 9         0.48860251190291987 * d.x,
10         1.0925484305920782  * d.x*d.y,
11         1.0925484305920782  * d.y*d.z,
12         0.94617469575755997 * (3.0*d.z*d.z-1.0),
13         1.0925484305920782  * d.z*d.x,
14         0.54627421529603904 * (d.x*d.x-d.y*d.y)
15     );
16 }
17 vec3 uvToDir(vec2 uv){
18     float phi   = (uv.x - 0.5) * 2.0 * 3.14159265;
19     float theta = (0.5 - uv.y) * 3.14159265;
20     float cp = cos(phi), sp = sin(phi);
21     float ct = cos(theta), st = sin(theta);
22     return vec3(ct * cp, st, ct * sp);
23 }
24 vec3 fresnelSchlick(float cosTheta, vec3 F0){
25     return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
26 }
27
28 void main(){
29     vec3 P  = texture(sTD2DInputs[0], vUV.st).rgb;
30     vec2 uv  = texture(sTD2DInputs[3], vUV.st).rg;
31     vec3 dir = uvToDir(uv);
32     vec3 Le  = texture(sTD2DInputs[1], uv).rgb;
33
34     float[9] Y = shBasis(dir);
35     vec3 coeff[9];
36     for (int i = 0; i < 9; ++i) coeff[i] = Y[i] * 0.1 * Le;
37
38     vec3 N0 = texture(sTD2DInputs[2], vUV.st).rgb;
39     vec3 N  = normalize(N0);
40     float[9] YN = shBasis(N);
41     vec3 shCol = vec3(0.0);
42     for (int i = 0; i < 9; ++i) shCol += YN[i] * coeff[i];
43
44     vec3 V0 = CamPos - P;
45     vec3 V   = normalize(V0);
46     vec3 F0  = mix(vec3(0.04), Le, 0.0);
47     vec3 F   = fresnelSchlick(max(dot(N, V), 0.0), F0);
48
49     vec3 envColor = shCol * (1.0 - F) + shCol * F;
50
51     fragColor = TDOutputSwizzle(vec4(envColor, 1.0));
52 }
```

Figure 28, Spherical harmonic function + Fresnel reflection glsl code;

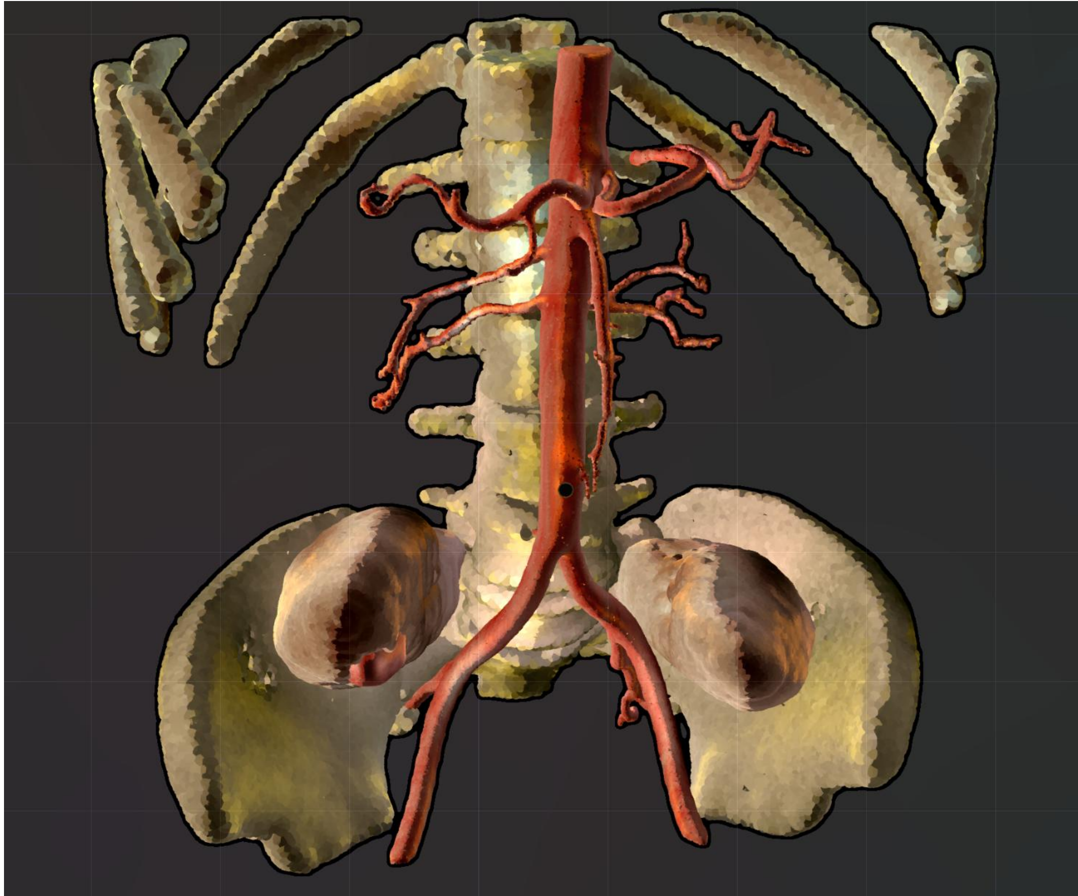

Figure 29, adjusted result,all sphere instances,bond for 100k particles,kidney for 30k particles,blood vessel;100k particles.The environment texture uses a colored noise map, Y_0^0 *= 0.6 in spherical harmonic functions.Perform convolution on the alpha channel of the final rendered image to obtain the edge areas, and then smooth the edges using blur and threshold methods.Among them, the shader of the blood vessels is connected to the a playing noise map, and a corresponding normal map is generated to simulate the blood flow visible from the outside.*

```glsl
out vec4 fragColor;

uniform vec3  uLightPos;
uniform float uRampWidth;
uniform float uShadowCut;
uniform float pow1;
uniform vec3 CamPos;
uniform float uMetallic;
uniform float NorPow;
uniform vec2 Till;

vec3 toonRamp(float ndl,float pow1){
    ndl *= pow1;
    float u = clamp(ndl, 0.0, 1.0);
    float edge  = smoothstep(uShadowCut - uRampWidth * 0.5,
                             uShadowCut + uRampWidth * 0.5, u);
    return texture(sTD2DInputs[2], vec2(edge, 0.5)).rgb;
}

vec3 viewNormWithZ(vec3 worldNorm, vec3 camPos, vec3 worldPos){
    vec3  V   = normalize(camPos - worldPos);
    vec3  f   = -V;
    vec3  up  = abs(f.y) < 0.999 ? vec3(0,1,0) : vec3(0,0,1);
    vec3  r   = normalize(cross(up, f));
    vec3  u   = cross(f, r);
    mat3  T   = mat3(r, u, f);
    return T * worldNorm;
}
float[9] shBasis(vec3 d){
    return float[9](
        0.58209479177387814,
        0.48860251190291987 * d.y,
        0.48860251190291987 * d.z,
        0.48860251190291987 * d.x,
        1.0925484305920782  * d.x*d.y,
        1.0925484305920782  * d.y*d.z,
        0.94617469575755997 * (3.0*d.z*d.z-1.0),
        1.0925484305920782  * d.z*d.x,
        0.54627421529603904 * (d.x*d.x-d.y*d.y)
    );
}
vec3 uvToDir(vec2 uv){
    float phi   = (uv.x - 0.5) * 2.0 * 3.14159265;
    float theta = (0.5 - uv.y) * 3.14159265;
    float cp = cos(phi), sp = sin(phi);
    float ct = cos(theta), st = sin(theta);
    return vec3(ct * cp, st, ct * sp);
}


void main(){
    vec2 UV = texture(sTD2DInputs[9], vUV.st).rg * Till;
    UV = UV - floor(UV);
    vec3 texPos = texture(sTD2DInputs[4], vUV.st).rgb;
    vec3 NO = normalize(texture(sTD2DInputs[0], vUV.st).rgb);

    vec3 ViewDir = normalize(CamPos - texPos);
    vec3 uLightDir = normalize(texPos - uLightPos);

    //ARM
    vec3 ARM = texture(sTD2DInputs[8], UV).rgb;
    float m = ARM.z;

    //TBN
    vec3 up = abs(NO.y) > 0.999 ? vec3(0,0,1) : vec3(0,1,0);
    vec3 T = normalize(cross(up, NO));
    vec3 B = cross(NO, T);
    mat3 TBN = mat3(T, B, NO);

    vec3 NorTex = texture(sTD2DInputs[7], UV).rgb * 2.0 - 1.0;
    vec3 WorldN = TBN * NorTex;
    vec3 FinalN = mix(NO,WorldN,NorPow);
    vec3 N = FinalN;
    vec3 vN = viewNormWithZ(N, CamPos, texPos);
    vec3 H = (uLightDir + ViewDir*-1)/2;
    float NoH = max(dot(H*-1,N),0.0) ;
    float NoV = dot(N,ViewDir) * -1;
    float NoL = max(dot(uLightDir,N)*-1,0.0);


    //HeightLight N . H
    float heiArea = step(0.9,NoH);
    float kS = mix(0.04, 1.0, uMetallic*m) * heiArea;
    float kD = (1.0 - kS) * (1.0 - uMetallic);
    vec3 heiCol = texture(sTD2DInputs[5], vUV.st).rgb * heiArea*kS;

    //BaseCol
    vec3 baseColor = texture(sTD2DInputs[1], UV).rgb;

    //Matcap
    vec3 McN = vN*0.5+0.5;
    vec3 McCol = texture(sTD2DInputs[3], McN.xy).rgb ;


    // 3. main light N·L
    float ndl = max(dot(N, normalize(uLightDir))*-1, 0.0);

    // 4. ramp
    vec3 rampColor = toonRamp(ndl,pow1);
    rampColor  *= kD;

    //environment light
    vec3 P2  = texture(sTD2DInputs[4], vUV.st).rgb;//pos
    vec2 uv2  = texture(sTD2DInputs[9], vUV.st).rg; //uv
    vec3 dir2 = uvToDir(uv2);
    vec3 Le2  = texture(sTD2DInputs[6], uv2).rgb; //map

    float[9] Y2 = shBasis(dir2);
    vec3 coeff[9];
    for (int i = 0; i < 9; ++i) coeff[i] = Y2[i] * 0.1 * Le2;

    vec3 N02 = texture(sTD2DInputs[0], vUV.st).rgb;//normal
    vec3 N2 = normalize(N02);
    float[9] YN2 = shBasis(N2);
    vec3 shCol2 = vec3(0.0);
    for (int i = 0; i < 9; ++i) shCol2 += YN2[i] * coeff[i];

    vec3 V02 = CamPos - P2;
    vec3 V2   = normalize(V02);
    vec3 F02  = mix(vec3(0.04), Le2, 0.0);
    float FFF = max(dot(N2, V2), 0.0);
    vec3 FS2   = FFF + (1.0 - FFF) * (1.0 - F02)*(1.0 - F02)*
    (1.0 - F02)*(1.0 - F02)*(1.0 - F02);

    vec3 envColor2 = shCol2 * (1.0 - F2) + shCol2 * FS2;

    //  final color
    vec3 finalRGB = baseColor * rampColor + McCol + heiCol + envColor2;

    //clip back
    vec3 V = vec3(ViewDir.x * -1,ViewDir.y,ViewDir.z * -1);
    float a = mix(0.5,1,step(0,dot(V,NO)));
    fragColor = TDOutputSwizzle(vec4(finalRGB, 1));
}
```

Figure 30, glsl code to the result above.

3.4 Particle animation setting

using the bounding box method: the first 2 nodes select the x-coordinate (i.e., the r value of the pixel) within the two extreme ranges of the position, the middle 2 nodes select the y-coordinate (i.e., the g value of the pixel) within the two extreme ranges of the position, and the last 2 nodes select the z-coordinate (i.e.,

the b value of the pixel) within the two extreme ranges of the position. Then, these ranges are combined through multiplication, and the selected particles move in the negative direction of the normal (multiply the texture storing the normal information by a coefficient and add it to the original texture storing the position information) to achieve the squeezing effect.Use the feedback node to record the results after compression (to prevent: compression occurs when in contact, and immediate restoration occurs when removed)

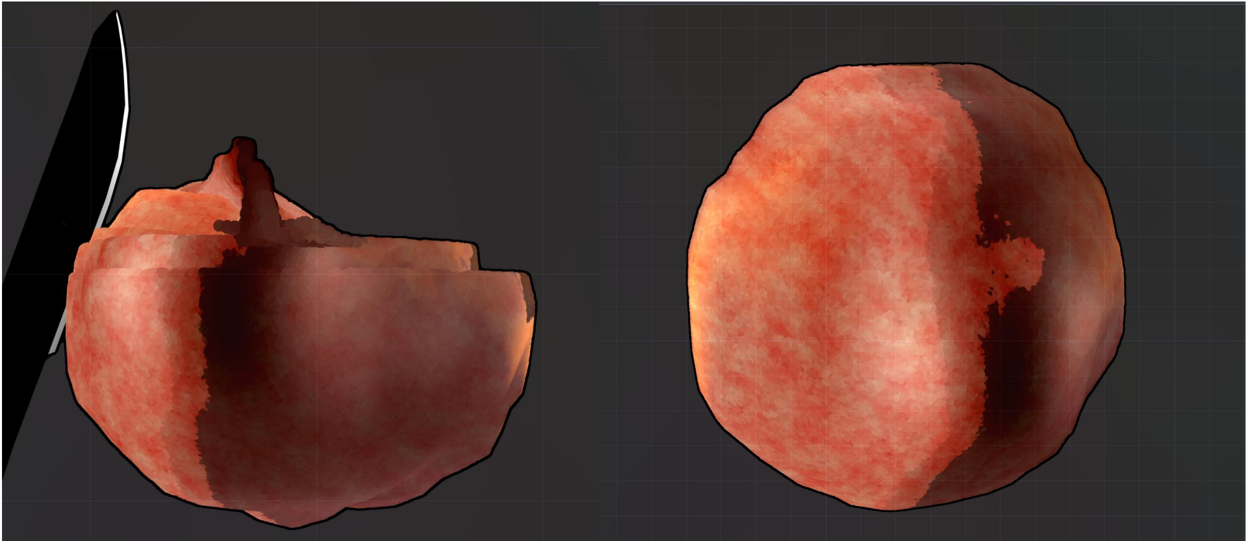

Figure 31, left:pressed tumour; right:origin tumour;

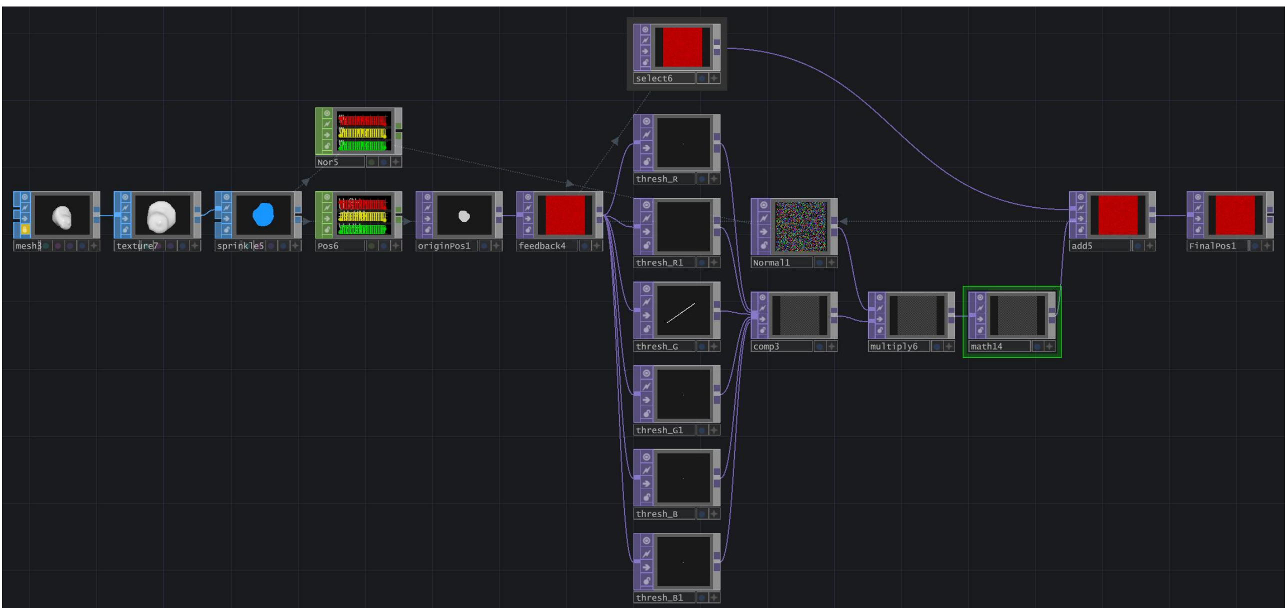

Figure 32,Nodes for calculating the surface compression effect

To simulate a stylized effect of blood flowing out, first, when generating particles, change it to generate particles within the volume of the mesh. Then, use the mouse position as the input for the x and y axes, the left mouse button as the input for the switch to test the interaction. The z-axis can be set to a constant, or use the analyze chop node to obtain the center position of the original vertices, take the z value of this position, and then calculate the distance between each point in the xy plane and the mouse position input ((x1-x2)**2+(y1-y2)**2). Set a thresh, filter out the vertices whose distance is within the range, and use the previously mentioned bounding box filtering method to select the vertices to be moved, and multiply the mouse left button input as the switch to determine which vertices have been interacted with. The vertex movement effect part uses simple noise multiplied by a coefficient to simulate a random effect, and adds a constant to simulate the effect of gravity. This changing value is

multiplied by a coefficient (this coefficient is controlled by the previous filtering method setting; the parts not selected are 0, that is, the position will not change, and the selected ones are set to around 0.03 to allow these points to change relatively smoothly). The feedback node records the position of the changed points to realistically show that the vertices updated in each frame are not the initial position vertices but the vertices that have been updated, that is, the particles move continuously over time. Then, use a pixel value ranging from 0 to 1 noise map, add a constant such as 0.05 to this map per frame, and use the thresh node to filter out the pixels with values greater than or equal to 1. Use the matte node to make these pixels with a value of 1 turn into (0, 0, 0), and increase them frame by frame to obtain a mask image that flickers according to the cycle. Then, use this image to multiply the texture storing the initial position and replace the current stored position texture to achieve the effect of making the particles return to the initial position after moving.

Figure 33, blood particle effect simulate.

Figure 34,Nodes for calculating the particle flow effect

3.4 Interaction connect

The Mediapipe plugin that can be used in TouchDesigner is encapsulated as Google's cross-platform

visual/audio pipeline framework. It can directly use a camera (such as the built-in camera of a laptop) as the input to track information such as a human's face, hands, and posture, and then use this information to achieve interaction. Here, hand tracking is used. The center position of one hand is passed in, and the x, y coordinates of this position are replaced with the x, y coordinates of the mouse position used during testing (since the effect of the test method, which uses Mediapipe to recognize the depth of the hand, was not obvious, it was discarded). Then, a preset gesture replaces the previous left mouse button of the mouse as a switch. Use the Select Node and Rename Node to change the names of the parameters to the same :”tx”,” ty”,” A”, as before to avoid conflicts. At the same time, add a surgical knife model and texture file. After the origin is set to 0, the mapped position information is passed to the position of the surgical knife. The effect of the gesture recognition button can also be connected to the rotation attribute of the surgical knife to enhance the visual effect of the interaction.

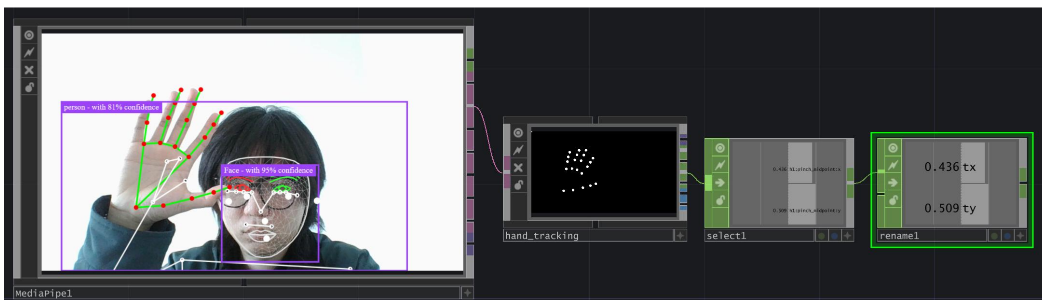

Figure 35, mediapiepe plugin tracking one hand,and get the position of midpoint on screen space.

4. _PSD Compare

4.1 visual effect compare

For users who have no background in medicine, it is difficult to understand the meaning of a set of CT images: which part of the body is abnormal or which part of the image corresponds to which part of the body.

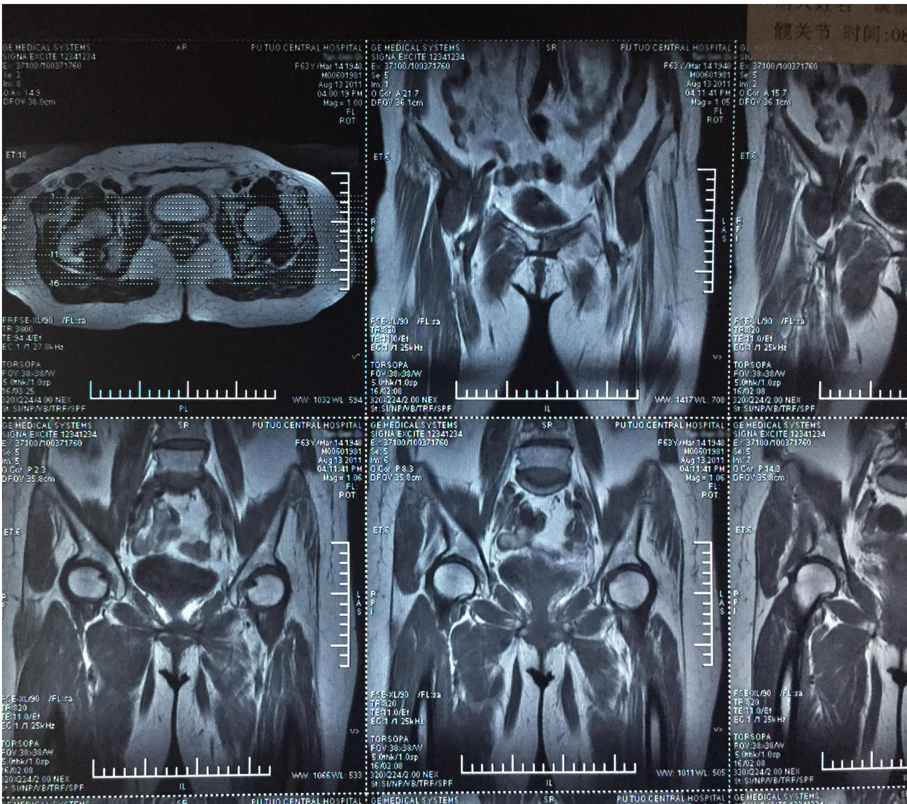

Figure 36. a set of CT images.

Realistic PBR materials are easy for solid static objects such as bones mesh rendering. They can use the texture blending method to draw blood. However, too many micro-surface details will make the picture look messy and difficult for the viewers to grasp the key points. Moreover, the interaction mode is stiff and is only suitable for teaching purposes for surgical operators. Patients may have difficulties understanding the surgical process.In this case, when dealing with kS, the setting might be too high, resulting in a reflection effect that is greater than the normal bone effect. Additionally, the main difference between the Unity material's PBR process and the traditional PBR process lies in how they handle the diffuse reflection results. In the traditional PBR process, to maintain energy conservation, the result of the Lambert model is divided by pi. However, in Unity's PBR process, in the diffuse reflection part, pi is not divided, but the lighting calculation result of the Lambert model is directly used. This might result in a slightly whiter appearance.

Figure 37. https://www.precisionostech.com/,The case diagram on the official website of PrecisionOS

Figure 38. https://www.syncvrmedical.com/,The case diagram on the official website of SyncVR Medical

Converting the model into particles and calculating the colors separately allows for more flexible control over the desired outcome. Besides using GLSL to adjust the lighting model to obtain the color results, parameters can also be set to control the image, such as using vertex IDs, normals, positions, UVs, etc. as inputs to adjust the color, size, position, rotation, scaling, or sampling different textures,or instance different units to achieve more visual effect expressions.

Figure 39. particles scaled by vertical position

4.2 Device compare

In VR devices, the common ones are: Xiaomi VR set,Meta Quest 2,Oculus Go 32 GB,pico3:approximately 200€;better ones: quest3,pico4,rokit,air,HTC approximately 600€;even better ones such as vision pro cost about 3000€. Moreover,each of these devices is for one person only. As teaching equipment,they are relatively expensive and not very good for people with myopia (wearing glasses often makes them tilt,each person needs to have their own special lenses,and currently,the lenses provided with the devices are mostly magnetic and prone to falling off).

The cost of the host + monitor solution can be controlled below 500eu. This not only enables it to accommodate more users simultaneously, but also is more friendly to users with myopia compared to VR devices. Moreover, in hospital environments, it can avoid cross-infection caused by contact (or it eliminates a large amount of disinfection workload).

The XR interaction mode recognized by the camera is also more direct. The hand recognition of VR devices can conduct interactions quite precisely, but it requires a lot of personalized adjustments in the early stage, such as eye distance, eye size, helmet tightness, etc. There are requirements for the surrounding open environment. Moreover, most VR devices will cause the image to appear distorted around the display screen when worn, and after long-term wearing, dizziness will occur. The current XR device like Quest 3 can project the scene in front of the user onto the user's eyes through the camera on the helmet in the image, but the resolution is not high. Especially when viewing the display screens of devices like mobile phones or computers, the experience is very poor. And the camera CMOS of VR devices is relatively small and has insufficient light sensitivity. It can hardly be used in dim environments. The combination of display and camera can avoid these shortcomings. In the evening, in some underdeveloped areas with dim hospital lighting, it can also be used normally.

5._PSD Conclusion

Currently, there are many users who need to acquire medical knowledge, such as patients visiting

hospitals, those who receive ct reports or examination forms and need to queue up to ask the doctor about the information contained therein. Moreover, most medical students lack effective training. The existing textbook images are mostly blurry and difficult to understand. Most current solutions mainly involve setting up some text and image materials, video materials on the web, or using VR devices to conduct surgical simulations, or using specific sensors to precisely read the position of the operator's hand for surgical simulations. The main drawbacks of these solutions include poor visual effects, limited user capacity, high cost, and difficulty in customization.
In this study, the CT files were used to generate mesh files, which were then input into the interactive software TouchDesigner. Particles were generated on the mesh surface for particle instances, GLSL shaders were written, and coloring was performed using the instance method. Additionally, the Mediapipe plugin was connected for camera interaction. Try to incorporate the cartoon rendering of NPR into the PBR process, and apply this workflow to medical education. This will enable the images to be expressed more concisely and clearly. Enable both doctors and patients to obtain the information in the document with a lower threshold.

References

Kim Seongdong.(2016).PBR(Physically based Render) simulation considered mathematical Fresnel model for Game Improvement

Lohre R, Bois A, Athwal GS, Lapner P, Pollock J, Goel DP.(2020).Effectiveness of Immersive Virtual Reality on Orthopedic Surgical Skills and Knowledge Acquisition Among Senior Surgical Residents

PBR theory,(Joey de Vries)Retrieved November 3, 2025 ,from https://learnopengl-cn.github.io/07%20PBR/01%20Theory/

Liang Wannian.(2004).中国社区卫生服务和全科医学教育的现状与对策[Status and Countermeasures of community health service and general medical education in China]

Cao Quanxi.(2024).A Study on Teaching Reform of the "Creative Coding" Course Based on TouchDesigner

Farhan Naseer;Vladimir Kazei;Weichang Li.(2024).Understanding 3D seismic data visualization with C++, OpenGL and GLSL

Sangkun Park.(2014).A Real-Time Rendering Algorithm of Large-Scale Point Clouds or Polygon Meshes Using GLSL

Lohre et al., 2020, Effectiveness of Immersive Virtual Reality on Orthopedic Surgical Skills and Knowledge Acquisition Among Senior Surgical Residents

Lohre & Goel, 2020, Immersive Virtual Reality: A Paradigm Shift in Education and Training

Lohre et al., 2020, Virtual Reality in Spinal Endoscopy: a Paradigm Shift in Education to Support Spine Surgeons

Nousiainen et al., 2018, Eight-year Outcomes of a Competency-Based Residency Training Program in Orthopedic Surgery

Lohre et al., 2020, Improved Complex Skill Acquisition by Immersive Virtual Reality

Lohre et al., 2020, Effectiveness of Immersive Virtual Reality on Orthopedic Surgical Skills and Knowledge Acquisition Among Senior Surgical Residents