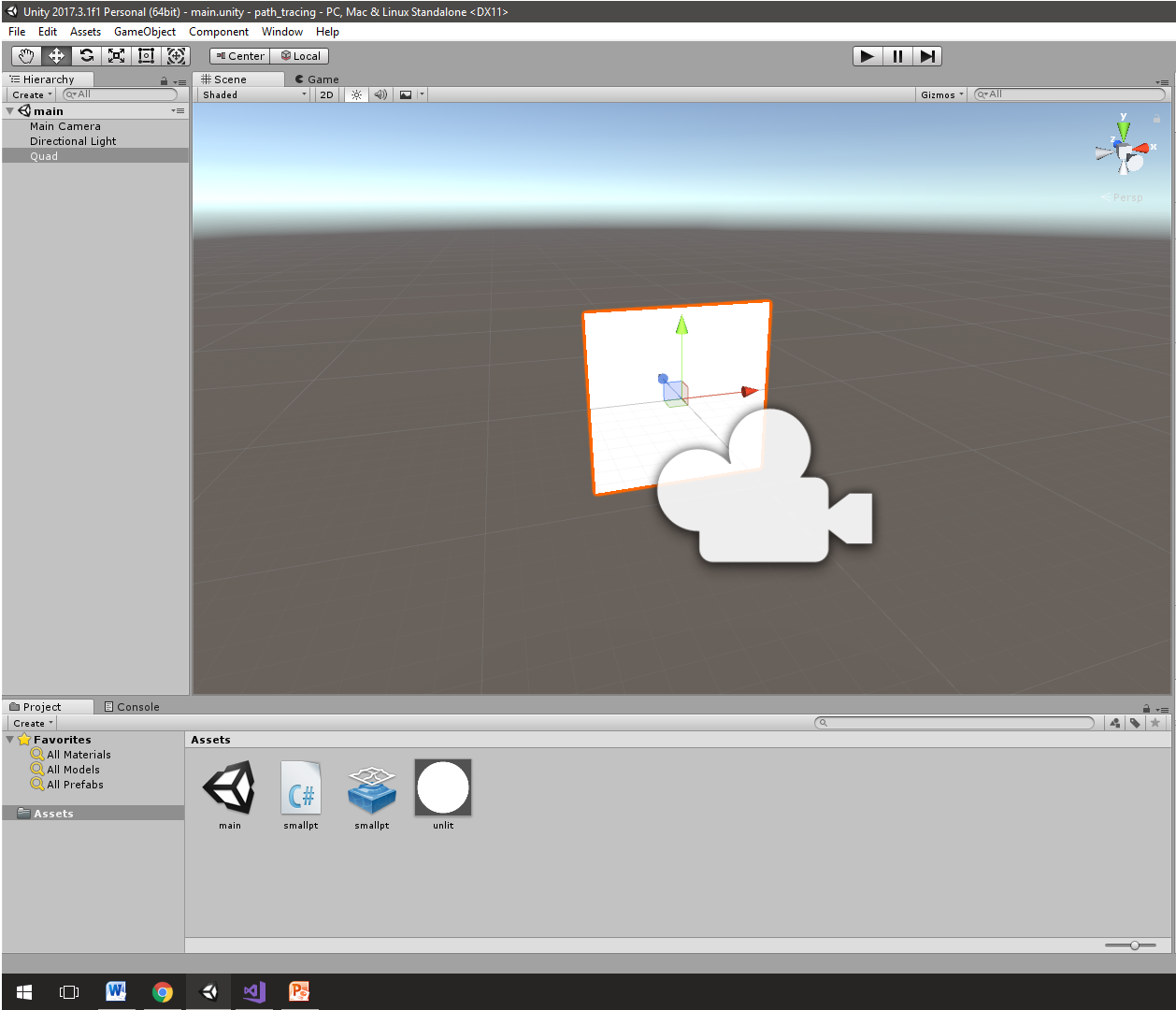
**Basic Scene**

The basic scene in unity consists of one Quad object with one C# script attached, which contains one shader variable and one material variable. Inside the script there are two rendertexture variables, both 1024x1024 in size. They are used as buffers for storing and retrieving data in both GPU and CPU side.



**C# Script**

In order to create the material, we use two rendertextures. During the execution of the update function these rendertexture buffers swap roles. In the beginning buffer A is used as a reader and B as a writer. Later, buffer B is used as a reader and buffer A as a writer. This way an average number of the colors for each pixels is calculated, which is used in monte carlo methods inside the shader.

using UnityEngine;

public class smallpt : MonoBehaviour{

public ComputeShader compute\_shader;

RenderTexture A, B;

public Material material;

int handle\_main;

int count = 0;

void Start(){

A = new RenderTexture(1024, 1024, 0);

A.enableRandomWrite = true;

A.Create();

B = new RenderTexture(1024, 1024, 0);

B.enableRandomWrite = true;

B.Create();

handle\_main = compute\_shader.FindKernel("CSMain");

}

void Update() {

compute\_shader.SetTexture(handle\_main, "reader", A);

compute\_shader.SetTexture(handle\_main, "writer", B);

compute\_shader.SetFloat("iFrame", count++);

compute\_shader.SetFloat("iTime", Time.time);

compute\_shader.Dispatch(handle\_main, A.width / 8, A.height / 8, 1);

compute\_shader.SetTexture(handle\_main, "reader", B);

compute\_shader.SetTexture(handle\_main, "writer", A);

compute\_shader.SetFloat("iFrame", count++);

compute\_shader.SetFloat("iTime", Time.time);

compute\_shader.Dispatch(handle\_main, B.width / 8, B.height / 8, 1);

material.mainTexture = B;

}  
}

**Compute Shader**

The shader works in this way. For each dispatch call, 1024 rays are cast in the scene (one for every pixel). Each ray intersects with objects ( Spheres only ) in order to accumulate the color for the specified pixel. A ray can bounce many times depending on the number of the MAXDEPTH variable. For every intersection, a new coordinate system is produced based on the up vector at the intersected point. Then a new ray is cast above the intersection point in random direction in hemisphere oriented around the up vector found.

Based on the sphere’s reflection type a ray can bounce in different ways:

* When a ray hits a diffuse surface, rays are cast at random angle above the hemisphere.
* When a ray hits a glass surface, the shader determines if it is entering or exiting the glass to compute the refraction ray.The dot product of ray and normal tell this. Glass is both reflective and refractive.
* When a ray hits a specular surface, the angle of incidence is equal to the glass of the reflection.

At the end, the resulting color is divided by the value of the iframe variable so as to produce an approximation of the samples.

**Code Breakdown**

Basic structs:

struct Ray

{

float3 o, d; // origin , direction

};

struct Sphere

{

float r; // radius

float3 p, e, c; // position, emission , color

int refl; // reflection type

};

Reflection types: Diffuse, Specular, Refractive

Light:

Sphere lightSourceVolume;

void initLight(){

lightSourceVolume.r = 20.0;

lightSourceVolume.p = float3(50.0, 81.6, 81.6);

lightSourceVolume.e = float3(12,12,12);

lightSourceVolume.c = float3(0,0,0);

lightSourceVolume.refl = DIFF;

}

Scene initialization ( Spheres + Walls ):

Sphere spheres[NUM\_SPHERES];

void initSpheres() {

spheres[0].r = 1e5;

spheres[0].p = float3(-1e5+1., 40.8, 81.6);

spheres[0].e = float3(0,0,0);

spheres[0].c = float3(.75, .25, .25);

spheres[0].refl = DIFF;

…

}

Intersections:  
  
Ray equation: P ( t ) = O + t\*D

The sphere equation: ( x - cx )2+( y - cy )2+( z - cz )2 + r2 = 0

In vector form: ( P - C ) \* ( P - C ) + r2 = 0

* ( O + t\*D – C )\*( O + t\*D – C ) – r2 = 0
* ( D\*D )\*t2 + 2\*D\*( O – C )\*t + ( O – C ) \* ( O – C ) – r2 = 0

Solutions:  
a = ( D\*D )   
  
b = 2\*D\*(O - C)

c = (O - C)\* (O - C) - r2t

Negative discriminant means that the ray missed the sphere.  
If both t values are negative, the sphere is behind the ray.

float intersect(Sphere s, Ray r) {

// p is sphere's center

float3 op = s.p - r.o;

// solutions of the sphere's quadratic formula

float t, epsilon = 1e-3, b = dot(op, r.d), det = b \* b - dot(op, op) + s.r \* s.r;

// ray misses sphere return nohit else return smaller postitive t

if (det < 0.) return 0.; else det = sqrt(det); return (t = b - det) > epsilon ? t : ((t = b + det) > epsilon ? t : 0.);

}

int intersect(Ray r, out float t, out Sphere s, int avoid) {

int id = -1;

t = 1e5;

s = spheres[0];

for (int i = 0; i < NUM\_SPHERES; ++i) {

Sphere S = spheres[i];

float d = intersect(S, r);

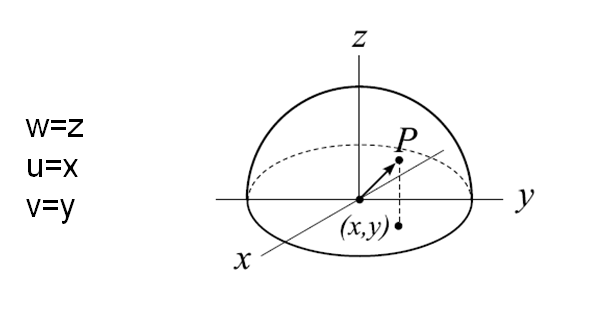
if (i!=avoid && d!=0. && d<t) { t = d; id = i; s=S; }

}

return id;

}

Sampling unit Hemisphere:



float3 jitter(float3 d, float phi, float sina, float cosa)

{

float3 w = normalize(d), u = normalize(cross(w.yzx, w)), v = cross(w, u);

return (u\*cos(phi) + v\*sin(phi)) \* sina + w \* cosa;

}

Radiance:

The MAXDEPTH variable is the number of bounces for each ray.

float3 radiance(Ray r)

{

float3 acc = float3(0,0,0);

float3 mask = float3(1,1,1);

int id = -1;

for (int depth = 0; depth < MAXDEPTH; ++depth)

{

float t;

Sphere obj;

// Check for intersections

if ((id = intersect(r, t, obj, id)) < 0) break;

// Create the up vector at intersected position

float3 x = t \* r.d + r.o;

float3 n = normalize(x - obj.p), nl = n \* sign(-dot(n, r.d));

if (obj.refl == DIFF) {

float r2 = rand();

float3 d = jitter(nl, 2.\*PI\*rand(), sqrt(r2), sqrt(1. - r2));

float3 e = float3(0,0,0);

Sphere s = lightSourceVolume;

int i = 8;

float3 l0 = s.p - x;

float cos\_a\_max = sqrt(1. - clamp(s.r \* s.r / dot(l0, l0), 0., 1.));

float cosa = lerp(cos\_a\_max, 1., rand());

float3 l = jitter(l0, 2.\*PI\*rand(), sqrt(1. - cosa\*cosa), cosa);

Ray ray1;

ray1.o=x;

ray1.d=l;

if (intersect(ray1, t, s, id) == i)

{

float omega = 2. \* PI \* (1. - cos\_a\_max);

e += (s.e \* clamp(dot(l, n),0.,1.) \* omega) / PI;

}

float E = 1.;

acc += mask \* obj.e \* E + mask \* obj.c \* e;

mask \*= obj.c;

Ray ray2;

ray2.o=x;

ray2.d=d;

r = ray2;

}

else if (obj.refl == SPEC)

{

acc += mask \* obj.e;

mask \*= obj.c;

Ray ray3;

ray3.o=x;

ray3.d=reflect(r.d, n);

r = ray3;

}

else

{

float a=dot(n,r.d), ddn=abs(a);

float nc=1., nt=1.5, nnt=lerp(nc/nt, nt/nc, float(a>0.));

float cos2t=1.-nnt\*nnt\*(1.-ddn\*ddn);

Ray ray4;

ray4.o=x;

ray4.d=reflect(r.d, n);

r = ray4;

if (cos2t>0.)

{

float3 tdir = normalize(r.d\*nnt + sign(a)\*n\*(ddn\*nnt+sqrt(cos2t)));

float R0=(nt-nc)\*(nt-nc)/((nt+nc)\*(nt+nc)),

c = 1.-lerp(ddn,dot(tdir, n),float(a>0.));

float Re=R0+(1.-R0)\*c\*c\*c\*c\*c,P=.25+.5\*Re,RP=Re/P,TP=(1.-Re)/(1.-P);

if (rand()<P)

{

mask \*= RP;

}

else

{

mask \*= obj.c\*TP;

Ray ray5;

ray5.o=x;

ray5.d=tdir;

r = ray5;

}

}

}

}

return acc;

}

Main:

[numthreads(8,8,1)]

void CSMain (uint2 id : SV\_DispatchThreadID)

{

float2 iResolution = float2(1024,1024);

float2 fragCoord = float2(id.x,id.y);

initLight();

initSpheres();

float2 st = fragCoord.xy / iResolution.xy;

seed = iTime+iResolution.y\*fragCoord.x/iResolution.x+fragCoord.y/iResolution.y;

float2 uv = 2. \* fragCoord.xy / iResolution.xy - 1.;

//Set up camera

float3 camPos = float3((2.\*.5\*iResolution.xy/iResolution.xy - 1.)\*float2(48., 40.)

+ float2(50., 40.8), 169.);

float3 cz = normalize(float3(50., 40., 81.6) - camPos);

float3 cx = float3(1., 0., 0.);

float3 cy = normalize(cross(cx, cz)); cx = cross(cz, cy);

float3 color = reader.SampleLevel(\_PointClamp,st,0).rgb\* float(iFrame);

//Initial ray

Ray ray6;

ray6.o=camPos;

ray6.d=normalize(.53135 \* (iResolution.x/iResolution.y\*uv.x\*cx + uv.y\*cy) + cz);

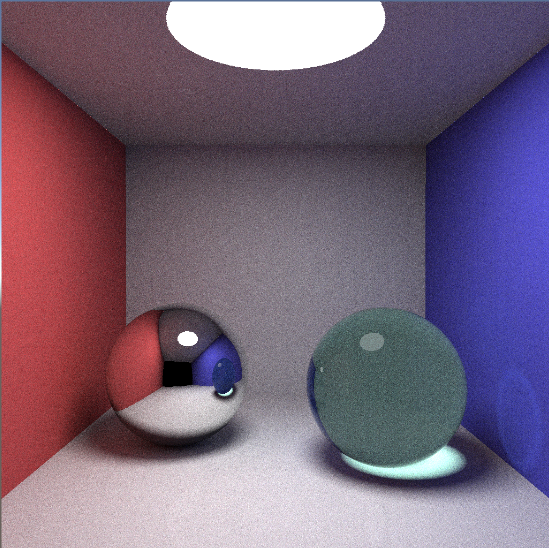
// Average color value

color += pow( abs(radiance(ray6)), float3(0.8, 0.8, 0.8))\*0.5;

writer[id] = float4(clamp(color/float(iFrame + 1.0),0.0,1.0),1.0);

}

**Final Output**



**Sources:**

http://www.kevinbeason.com/smallpt/  
https://www.shadertoy.com/view/4dtGWB