

INFOGR – Computer Graphics

J. Bikker - April-July 2015 - Lecture 12: “Advanced Shading”

Welcome!



Today's Agenda:

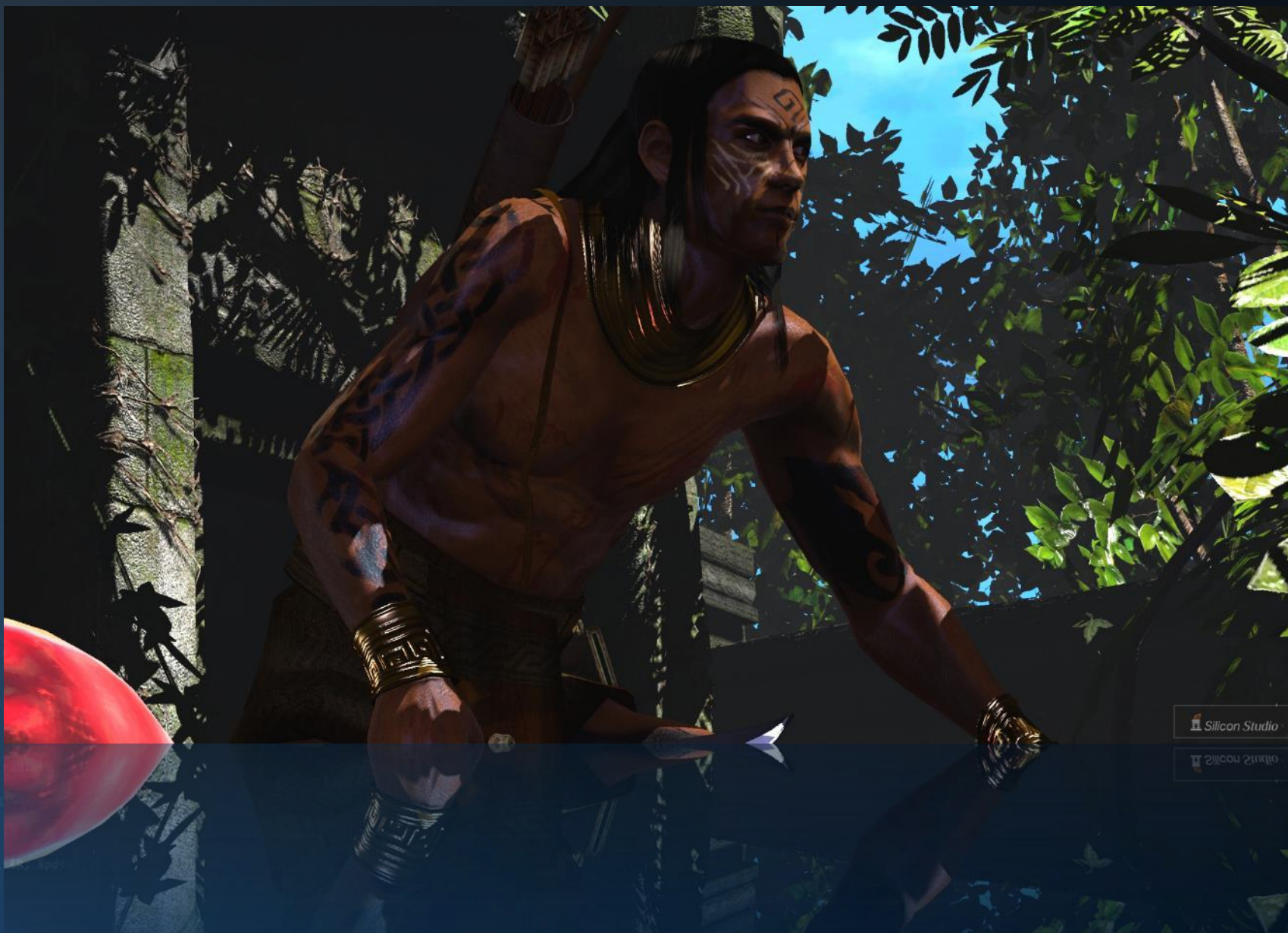
- The Postprocessing Pipeline
 - Vignetting, Chromatic Aberration
 - Film Grain
 - HDR effects
 - Color Grading
 - Depth of Field
- Screen Space Algorithms
 - Ambient Occlusion
 - Screen Space Reflections



```

rics
& (depth < MAXDEPTH)
{
    t = inside / 1.5;
    nt = nt / nc; dde = dde / nc;
    cos2t = 1.0f - nnt; r = sqrt(
    D, N );
    )
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (RB + (1 - RB) * r);
        Tr) R = (D * nnt - N * (dde
    )
    E * diffuse;
    = true;
    }
    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse,
    estimation - doing it properly, closely
    if;
    radiance = SampleLight( &rand, I, &t, &align
    e.x + radiance.y + radiance.z) > 0) && (refl
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurv;
    at3 factor = diffuse * INVPI;
    at weight = Mix2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (
    random walk - done properly, closely following
    ve)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &align
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}

```



Introduction

Post Processing

Operations carried out on a rendered image.

Purposes:

- Simulation of camera effects
- Simulation of the effects of HDR
- Artistic tweaking of look and feel, separate from actual rendering
- Calculating light transport in space
- Anti-aliasing

Post processing is handled by the *post processing pipeline*.

Input: rendered image, in linear color format;

Output: image ready to be displayed on the monitor.



Camera Effects

Purpose: simulating camera / sensor behavior

Bright lights:

- Lens flares
- Glow
- Exposure adjustment
- Trailing / ghosting



Camera Effects

Purpose: simulating camera / sensor behavior

Camera imperfections:

- Vignetting
- Chromatic aberration
- Noise / grain



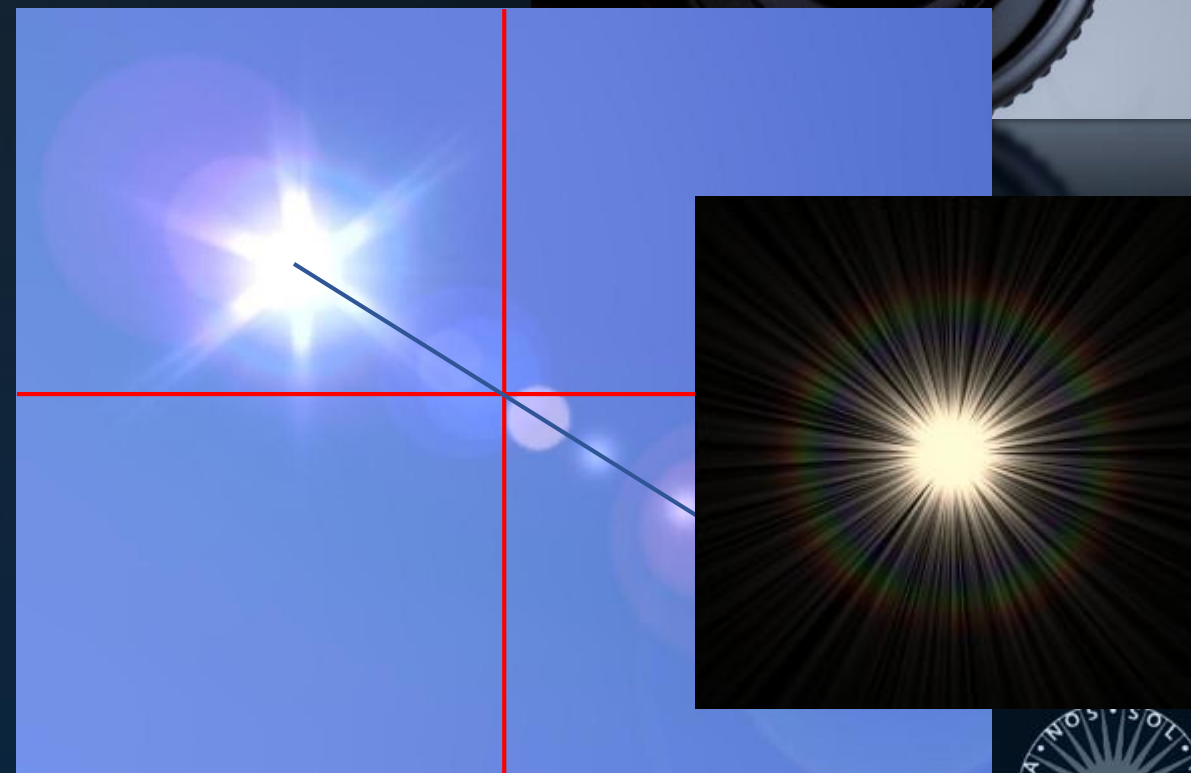
Camera Effects

Lens Flares

Lens flares are the result of reflections in the camera lens system.

Lens flares are typically implemented by drawing sprites, along a line through the center of the screen, with translucency relative to the brightness of the light source.

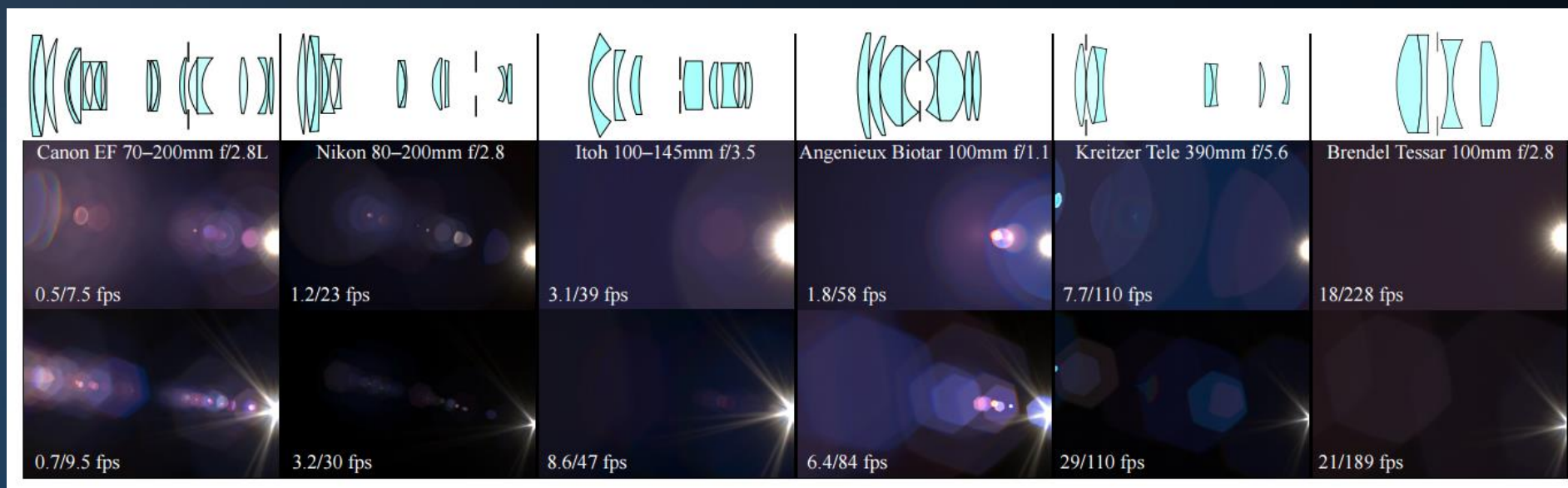
Notice that this type of lens flare is specific to cameras; the human eye has a drastically different response to bright lights.



Camera Effects

Lens Flares

“Physically-Based Real-Time Lens Flare Rendering”, Hullin et al., 2011





Camera Effects

Lens Flares



From: www.alienscribbleinteractive.com/Tutorials/lens_flare_tutorial.html



Vignetting

Cheap cameras often suffer from vignetting: reduced brightness of the image for pixels further away from the center.





SELF STORAGE


```

    rics
    & (depth < MAXDEPTH)
    {
        nt = inside / 1.5;
        nt = nt / nc;
        cos2t = 1.0f - nnt;
        D, N );
    }

    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (RB + (1 - RB) * a);
    Tr) R = (D * nnt - N * (a *

    E * diffuse;
    = true;

    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;

    MAXDEPTH)

    survive = SurvivalProbability( diffuse
    estimation - doing it properly, close
    if;
    radiance = SampleLight( &rand, I, &L,
    e.x + radiance.y + radiance.z) > 0) &

    v = true;
    at brdfPdf = EvaluateDiffuse( L, N );
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / direct

    random walk - done properly, closely f
    rive)

    ;
    at3 brdf = SampleDiffuse( diffuse, N,
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```





Camera Effects

Vignetting

Cheap cameras often suffer from vignetting: reduced brightness of the image for pixels further away from the center.

In a renderer, subtle vignetting can add to the mood of a scene.

Vignetting is simple to implement: just darken the output based on the distance to the center of the screen.



Camera Effects

Chromatic Aberration

This is another effect known from cheap cameras.

A camera may have problems keeping colors for a pixel together, especially near the edges of the image.

In this screenshot (from “Colonial Marines”, a CryEngine game), the effect is used to suggest player damage.





0 11:05 0

0



+100 0

9

10



Use a slightly different distance from the center of the screen when reading red, green and blue.

```

    brdf = SampleDiffuse( diffuse,
                           survive;
                           pdf;
                           E * brdf * (dot( N, R ) / pdf);
    return true;
}

```



Noise / Grain

Adding (on purpose) some noise to the rendered image can further emphasize the illusion of watching a movie.

```
pdf;
h = E * brdf * (dot( N, R ) / pdf);
```





Blair witch project



Camera Effects

Noise / Grain

Adding (on purpose) some noise to the rendered image can further emphasize the illusion of watching a movie.

Film grain is generally not static and changes every frame. A random number generator lets you easily add this effect (keep it subtle!).

When done right, some noise reduces the ‘cleanness’ of a rendered image.



Today's Agenda:

- The Postprocessing Pipeline
 - Vignetting, Chromatic Aberration
 - Film Grain
 - HDR effects
 - Color Grading
 - Depth of Field
- Screen Space Algorithms
 - Ambient Occlusion
 - Screen Space Reflections




```

1000 (depth < MAXDEPTH)
1001 {
1002     // Sample a point on the surface of the sphere
1003     Vec w = inside / (1 + sqrt(1 - inside));
1004     Vec nt = nt / nc, ddn = ddn / nc;
1005     Vec p2t = 1.0f - nnt * ddn;
1006     Vec D, N };
1007     Vec R = (D * p2t + N * ddn) * nt;
1008     Vec a = nt - nc, b = nt + nc;
1009     Vec Tr = 1 - (RR + (1 - RR) * Vec(0.25f));
1010     Vec R = (D * nnt - N * ddn) * Tr;
1011     Vec E * diffuse;
1012     Vec R = Tr * R;
1013     Vec refl = refl * R;
1014     Vec refl + refr)) && (depth < MAXDEPTH)
1015     {
1016         Vec D, N };
1017         Vec refl * E * diffuse;
1018         Vec R = Tr * R;
1019         Vec R = (D * nnt - N * ddn) * Tr;
1020         Vec E * diffuse;
1021         Vec R = Tr * R;
1022         Vec refl = refl * R;
1023         Vec refl + refr)) && (depth < MAXDEPTH)
1024     {
1025         Vec survive = SurvivalProbability( diffuse );
1026         Vec estimation - doing it properly, closely following survival;
1027         Vec r1, r2, &R, &pdf;
1028         Vec radiance = SampleLight( &rand, I, &L, &align, &pdf );
1029         Vec e.x + radiance.y + radiance.z) > 0) && (cosThetaOut > 0)
1030         {
1031             Vec w = true;
1032             Vec brdfPdf = EvaluateDiffuse( L, N ) * Psurf(x);
1033             Vec st3 factor = diffuse * INVPI;
1034             Vec st3 weight = Mis2( directPdf, brdfPdf );
1035             Vec st3 cosThetaOut = dot( N, L );
1036             Vec E * ((weight * cosThetaOut) / directPdf) * (radiance);
1037             Vec random walk - done properly, closely following survival;
1038             Vec survive);
1039             Vec st3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
1040             Vec survive;
1041             Vec pdf;
1042             Vec n = E * brdf * (dot( N, R ) / pdf);
1043             Vec sion = true;
1044             Vec survive;
1045             Vec pdf;
1046             Vec n = E * brdf * (dot( N, R ) / pdf);
1047             Vec sion = true;
1048             Vec survive;
1049             Vec pdf;
1050             Vec n = E * brdf * (dot( N, R ) / pdf);
1051             Vec sion = true;
1052             Vec survive;
1053             Vec pdf;
1054             Vec n = E * brdf * (dot( N, R ) / pdf);
1055             Vec sion = true;
1056             Vec survive;
1057             Vec pdf;
1058             Vec n = E * brdf * (dot( N, R ) / pdf);
1059             Vec sion = true;
1060             Vec survive;
1061             Vec pdf;
1062             Vec n = E * brdf * (dot( N, R ) / pdf);
1063             Vec sion = true;
1064             Vec survive;
1065             Vec pdf;
1066             Vec n = E * brdf * (dot( N, R ) / pdf);
1067             Vec sion = true;
1068             Vec survive;
1069             Vec pdf;
1070             Vec n = E * brdf * (dot( N, R ) / pdf);
1071             Vec sion = true;
1072             Vec survive;
1073             Vec pdf;
1074             Vec n = E * brdf * (dot( N, R ) / pdf);
1075             Vec sion = true;
1076             Vec survive;
1077             Vec pdf;
1078             Vec n = E * brdf * (dot( N, R ) / pdf);
1079             Vec sion = true;
1080             Vec survive;
1081             Vec pdf;
1082             Vec n = E * brdf * (dot( N, R ) / pdf);
1083             Vec sion = true;
1084             Vec survive;
1085             Vec pdf;
1086             Vec n = E * brdf * (dot( N, R ) / pdf);
1087             Vec sion = true;
1088             Vec survive;
1089             Vec pdf;
1090             Vec n = E * brdf * (dot( N, R ) / pdf);
1091             Vec sion = true;
1092             Vec survive;
1093             Vec pdf;
1094             Vec n = E * brdf * (dot( N, R ) / pdf);
1095             Vec sion = true;
1096             Vec survive;
1097             Vec pdf;
1098             Vec n = E * brdf * (dot( N, R ) / pdf);
1099             Vec sion = true;
1100             Vec survive;
1101             Vec pdf;
1102             Vec n = E * brdf * (dot( N, R ) / pdf);
1103             Vec sion = true;
1104             Vec survive;
1105             Vec pdf;
1106             Vec n = E * brdf * (dot( N, R ) / pdf);
1107             Vec sion = true;
1108             Vec survive;
1109             Vec pdf;
1110             Vec n = E * brdf * (dot( N, R ) / pdf);
1111             Vec sion = true;
1112             Vec survive;
1113             Vec pdf;
1114             Vec n = E * brdf * (dot( N, R ) / pdf);
1115             Vec sion = true;
1116             Vec survive;
1117             Vec pdf;
1118             Vec n = E * brdf * (dot( N, R ) / pdf);
1119             Vec sion = true;
1120             Vec survive;
1121             Vec pdf;
1122             Vec n = E * brdf * (dot( N, R ) / pdf);
1123             Vec sion = true;
1124             Vec survive;
1125             Vec pdf;
1126             Vec n = E * brdf * (dot( N, R ) / pdf);
1127             Vec sion = true;
1128             Vec survive;
1129             Vec pdf;
1130             Vec n = E * brdf * (dot( N, R ) / pdf);
1131             Vec sion = true;
1132             Vec survive;
1133             Vec pdf;
1134             Vec n = E * brdf * (dot( N, R ) / pdf);
1135             Vec sion = true;
1136             Vec survive;
1137             Vec pdf;
1138             Vec n = E * brdf * (dot( N, R ) / pdf);
1139             Vec sion = true;
1140             Vec survive;
1141             Vec pdf;
1142             Vec n = E * brdf * (dot( N, R ) / pdf);
1143             Vec sion = true;
1144             Vec survive;
1145             Vec pdf;
1146             Vec n = E * brdf * (dot( N, R ) / pdf);
1147             Vec sion = true;
1148             Vec survive;
1149             Vec pdf;
1150             Vec n = E * brdf * (dot( N, R ) / pdf);
1151             Vec sion = true;
1152             Vec survive;
1153             Vec pdf;
1154             Vec n = E * brdf * (dot( N, R ) / pdf);
1155             Vec sion = true;
1156             Vec survive;
1157             Vec pdf;
1158             Vec n = E * brdf * (dot( N, R ) / pdf);
1159             Vec sion = true;
1160             Vec survive;
1161             Vec pdf;
1162             Vec n = E * brdf * (dot( N, R ) / pdf);
1163             Vec sion = true;
1164             Vec survive;
1165             Vec pdf;
1166             Vec n = E * brdf * (dot( N, R ) / pdf);
1167             Vec sion = true;
1168             Vec survive;
1169             Vec pdf;
1170             Vec n = E * brdf * (dot( N, R ) / pdf);
1171             Vec sion = true;
1172             Vec survive;
1173             Vec pdf;
1174             Vec n = E * brdf * (dot( N, R ) / pdf);
1175             Vec sion = true;
1176             Vec survive;
1177             Vec pdf;
1178             Vec n = E * brdf * (dot( N, R ) / pdf);
1179             Vec sion = true;
1180             Vec survive;
1181             Vec pdf;
1182             Vec n = E * brdf * (dot( N, R ) / pdf);
1183             Vec sion = true;
1184             Vec survive;
1185             Vec pdf;
1186             Vec n = E * brdf * (dot( N, R ) / pdf);
1187             Vec sion = true;
1188             Vec survive;
1189             Vec pdf;
1190             Vec n = E * brdf * (dot( N, R ) / pdf);
1191             Vec sion = true;
1192             Vec survive;
1193             Vec pdf;
1194             Vec n = E * brdf * (dot( N, R ) / pdf);
1195             Vec sion = true;
1196             Vec survive;
1197             Vec pdf;
1198             Vec n = E * brdf * (dot( N, R ) / pdf);
1199             Vec sion = true;
1200             Vec survive;
1201             Vec pdf;
1202             Vec n = E * brdf * (dot( N, R ) / pdf);
1203             Vec sion = true;
1204             Vec survive;
1205             Vec pdf;
1206             Vec n = E * brdf * (dot( N, R ) / pdf);
1207             Vec sion = true;
1208             Vec survive;
1209             Vec pdf;
1210             Vec n = E * brdf * (dot( N, R ) / pdf);
1211             Vec sion = true;
1212             Vec survive;
1213             Vec pdf;
1214             Vec n = E * brdf * (dot( N, R ) / pdf);
1215             Vec sion = true;
1216             Vec survive;
1217             Vec pdf;
1218             Vec n = E * brdf * (dot( N, R ) / pdf);
1219             Vec sion = true;
1220             Vec survive;
1221             Vec pdf;
1222             Vec n = E * brdf * (dot( N, R ) / pdf);
1223             Vec sion = true;
1224             Vec survive;
1225             Vec pdf;
1226             Vec n = E * brdf * (dot( N, R ) / pdf);
1227             Vec sion = true;
1228             Vec survive;
1229             Vec pdf;
1230             Vec n = E * brdf * (dot( N, R ) / pdf);
1231             Vec sion = true;
1232             Vec survive;
1233             Vec pdf;
1234             Vec n = E * brdf * (dot( N, R ) / pdf);
1235             Vec sion = true;
1236             Vec survive;
1237             Vec pdf;
1238             Vec n = E * brdf * (dot( N, R ) / pdf);
1239             Vec sion = true;
1240             Vec survive;
1241             Vec pdf;
1242             Vec n = E * brdf * (dot( N, R ) / pdf);
1243             Vec sion = true;
1244             Vec survive;
1245             Vec pdf;
1246             Vec n = E * brdf * (dot( N, R ) / pdf);
1247             Vec sion = true;
1248             Vec survive;
1249             Vec pdf;
1250             Vec n = E * brdf * (dot( N, R ) / pdf);
1251             Vec sion = true;
1252             Vec survive;
1253             Vec pdf;
1254             Vec n = E * brdf * (dot( N, R ) / pdf);
1255             Vec sion = true;
1256             Vec survive;
1257             Vec pdf;
1258             Vec n = E * brdf * (dot( N, R ) / pdf);
1259             Vec sion = true;
1260             Vec survive;
1261             Vec pdf;
1262             Vec n = E * brdf * (dot( N, R ) / pdf);
1263             Vec sion = true;
1264             Vec survive;
1265             Vec pdf;
1266             Vec n = E * brdf * (dot( N, R ) / pdf);
1267             Vec sion = true;
1268             Vec survive;
1269             Vec pdf;
1270             Vec n = E * brdf * (dot( N, R ) / pdf);
1271             Vec sion = true;
1272             Vec survive;
1273             Vec pdf;
1274             Vec n = E * brdf * (dot( N, R ) / pdf);
1275             Vec sion = true;
1276             Vec survive;
1277             Vec pdf;
1278             Vec n = E * brdf * (dot( N, R ) / pdf);
1279             Vec sion = true;
1280             Vec survive;
1281             Vec pdf;
1282             Vec n = E * brdf * (dot( N, R ) / pdf);
1283             Vec sion = true;
1284             Vec survive;
1285             Vec pdf;
1286             Vec n = E * brdf * (dot( N, R ) / pdf);
1287             Vec sion = true;
1288             Vec survive;
1289             Vec pdf;
1290             Vec n = E * brdf *
```





HDR

HDR Bloom

A monitor generally does not directly display HDR images. To suggest brightness, we use hints that our eyes interpret as the result of bright lights:

- Flares
- Glow
- Exposure control



HDR

HDR Bloom

Calculation of HDR bloom:

1. For each pixel, subtract (1,1,1) and clamp to 0 (this yields an image with only the bright pixels)
2. Apply a Gaussian blur to this buffer
3. Add the result to the original frame buffer.





Unreal Engine 4

```

    (depth < MAXDEPTH)
    {
        // Inside / Left hemisphere
        nt = inside / 1.5; // 1.5 = 1 + 0.5
        nc = nt * (1 + nt); // ddx = 1 + ddx
        nt = nt / nc; ddx = ddx / nc;
        cos2t = 1.0f - nnt * wnt;
        D, N );
    }
}

// Inside / Right hemisphere
// at a = nt - nc, b = nt * nc
// at Tr = 1 - (RB + (1 - RB) * cos2t)
// Tr) R = (D * nnt - N * (ddx
//
// E * diffuse;
// = true;
//
//
// refl + refr)) && (depth < MAXDEPTH)
//
// D, N );
// refl * E * diffuse;
// = true;
//
//
// MAXDEPTH)
//
// survive = SurvivalProbability( diffuse );
// estimation - doing it properly, closely following
// df;
// radiance = SampleLight( &rand, I, &L, &align,
// .x + radiance.y + radiance.z) > 0) && (cos2t <
//
// w = true;
// at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
// at3 factor = diffuse * INVPI;
// at weight = Mix2( directPdf, brdfPdf );
// at cosThetaOut = dot( N, L );
// E * ((weight * cosThetaOut) / directPdf) * (rad
//
// random walk - done properly, closely following
// (survive)
//
//
// at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R
// survive;
// pdf;
// n = E * brdf * (dot( N, R ) / pdf);
// sion = true;

```

```

        int = nt / nc, ddc = ddc + 1;
        pos2t = 1.0f - nnt * 0.5f;
        D, N );
        0);
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (RB + (1 - RB) * pos2t);
        Tr) R = (D * nnt - N * (pos2t
        E * diffuse;
        = true;
        -
        efl + refr)) && (depth < MAXDEPTH)
        D, N );
        efl * E * diffuse;
        = true;
        MAXDEPTH)
        survive = SurvivalProbability( diffuse, N,
        estimation - doing it properly, closely following
        df;
        radiance = SampleLight( &rand, I, &L, &light,
        e.x + radiance.y + radiance.z) > 0) && (maxN
        v = true;
        at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        at3 factor = diffuse * INVPI;
        at weight = Mis2( directPdf, brdfPdf );
        at cosThetaOut = dot( N, L );
        E * ((weight * cosThetaOut) / directPdf) * (pdf
        random walk - done properly, closely following the
        ivate)
        ;
        at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R
        survive;
        pdf;
        n = E * brdf * (dot( N, R ) / pdf);
        sion = true;

```

```

a = nt - r; b = nt - r;
at Tr = 1 - (R0 + (1 - R0) * cosThetaOut);
Tr) R = (D * nnt - N * cosThetaOut) * Tr;

E * diffuse;
= true;

-
refl + refr)) && (depth < MAXDEPTH);

D, N );
-refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse, N );
estimation - doing it properly, clearly not
ff;
radiance = SampleLight( &rand, I, &t, &light,
2.x + radiance.y + radiance.z) > 0) && (max <
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mix2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * Pdf;
random walk - done properly, closely following
vive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R,
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

- ```

refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability(diffuse, L, N);
estimation - doing it properly, closely following a
ff;
radiance = SampleLight(@rand, I, @t, @lightType,
r.x + radiance.y + radiance.z) > 0) && (max < 0.
w = true;
at brdfPdf = EvaluateDiffuse(L, N) * Psurf;
at3 factor = diffuse * INVPI;
at weight = Mix2(directPdf, brdfPdf);
at cosThetaOut = dot(N, L);
E * ((weight * cosThetaOut) / directPdf) * (rad
random walk - done properly, closely following a
vive)

;
at3 brdf = SampleDiffuse(diffuse, N, r1, r2, @R
survive;
pdf;
n = E * brdf * (dot(N, R) / pdf);
sion = true;

```

```

survive = SurvivalProbability(diffuse, L, N);
// estimation - doing it properly, closely following the
// diff;
radiance = SampleLight(&rand, L, &L, &align, &align,
x.x + radiance.y + radiance.z) > 0) && (rand.y <
// w = true;
// at brdfPdf = EvaluateDiffuse(L, N); * Psurvive;
// at3 factor = diffuse * INVPI;
// at weight = Mix2(directPdf, brdfPdf);
// at cosThetaOut = dot(N, L);
// E * ((weight * cosThetaOut) / directPdf) * (rad
// random walk - done properly, closely following the
// survive)
//
// at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R
// survive;
// pdf;
// n = E * brdf * (dot(N, R) / pdf);
// nion = true;

```

```

 brdfPdf = evaluateDiffuse(L, N) * Psum;
 at3 factor = diffuse * INVPI;
 at weight = Mis2(directPdf, brdfPdf);
 at cosThetaOut = dot(N, L);
 E * ((weight * cosThetaOut) / directPdf) * (rad
random walk - done properly, closely following
ive)

;
at3 brdf = SampleDiffuse(diffuse, N, r1, r2, R
urview;
pdf;
n = E * brdf * (dot(N, R) / pdf);
sion = true;

```













# Today's Agenda:

- The Postprocessing Pipeline
  - Vignetting, Chromatic Aberration
  - Film Grain
  - HDR effects
  - Color Grading
  - Depth of Field
- Screen Space Algorithms
  - Ambient Occlusion
  - Screen Space Reflections



# Color Grading

## Color Correction

Changing the color scheme of a scene can dramatically affect the mood.

(in the following movie, notice how often the result ends up emphasizing blue and orange)

```

 if (depth < MAXDEPTH)
 {
 // Inside the sphere
 Vec r = inside / 1.5;
 Vec nt = nt / nc; nnt = nnt * nnt;
 Vec os2t = 1.0f - nnt; // weight for diffuse
 Vec D, N);
 }

 // Russian roulette
 Vec a = nt - nc, b = nt + nc;
 Vec Tr = 1 - (RR + (1 - RR) * r);
 Vec R = (D * nnt - N * (1 - nnt));

 Vec E * diffuse;
 bool = true;

 // Russian roulette
 Vec refl + refr)) && (depth < MAXDEPTH)
 {
 Vec D, N);
 Vec refl * E * diffuse;
 bool = true;

 }

 if (depth < MAXDEPTH)
 {
 Vec survive = SurvivalProbability(diffuse);
 // Russian roulette - doing it properly, closely following survival
 if (survive)
 {
 Vec radiance = SampleLight(&rand, I, &L, &light);
 Vec e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
 {
 Vec v = true;
 Vec brdfPdf = EvaluateDiffuse(L, N) * Psurvive;
 Vec t3 factor = diffuse * INVPI;
 Vec weight = Mis2(directPdf, brdfPdf);
 Vec cosThetaOut = dot(N, L);
 Vec E * ((weight * cosThetaOut) / directPdf) * (radiance.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
 }

 // Russian roulette - done properly, closely following survival
 Vec survive)
 {
 Vec t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf);
 Vec survive;
 Vec pdf;
 Vec n = E * brdf * (dot(N, R) / pdf);
 Vec sion = true;
 }
 }
 }

```







# Color Grading

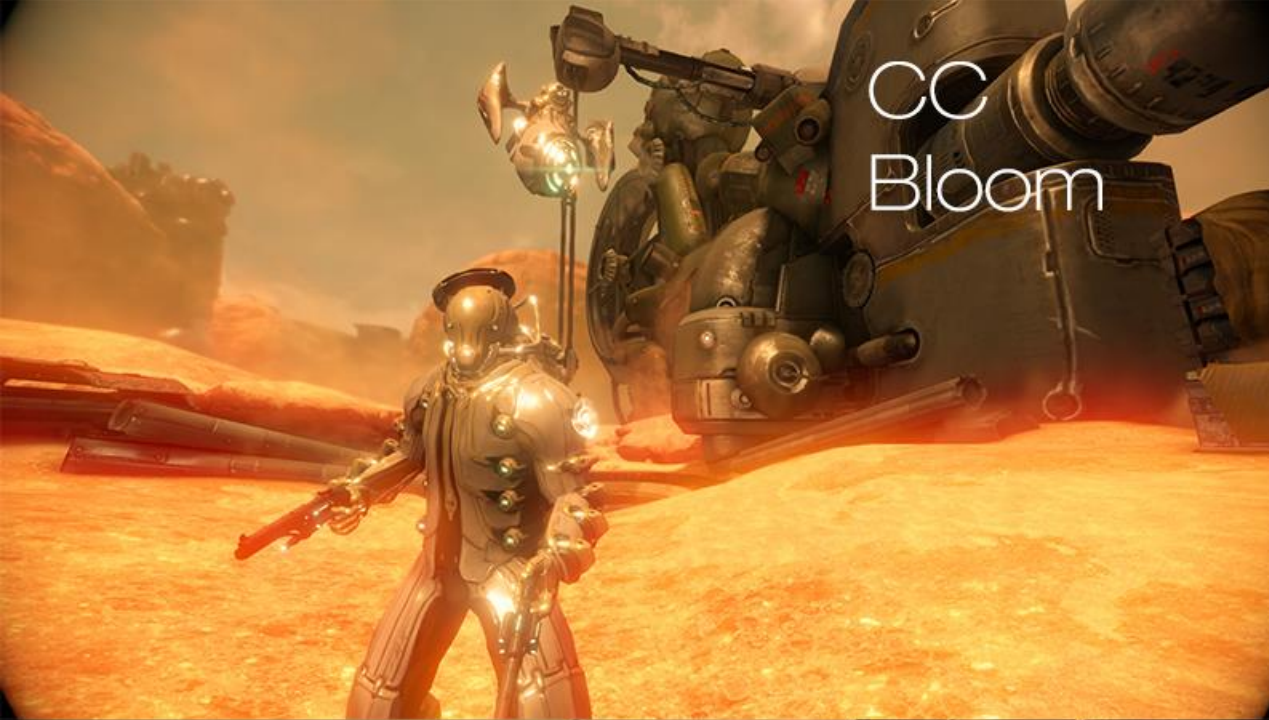
## Color Correction

### Color correction in a real-time engine:

1. Take a screenshot from within your game
2. Add a color cube to the image
3. Load the image in Photoshop
4. Apply color correction until desired result is achieved
5. Extract modified color cube
6. Use modified color cube to lookup colors at runtime.

















# Today's Agenda:

- The Postprocessing Pipeline
  - Vignetting, Chromatic Aberration
  - Film Grain
  - HDR effects
  - Color Grading
  - Depth of Field
- Screen Space Algorithms
  - Ambient Occlusion
  - Screen Space Reflections





# Gamma Correction

## Concept



Monitors respond in a non-linear fashion to input.



# Gamma Correction

## Concept

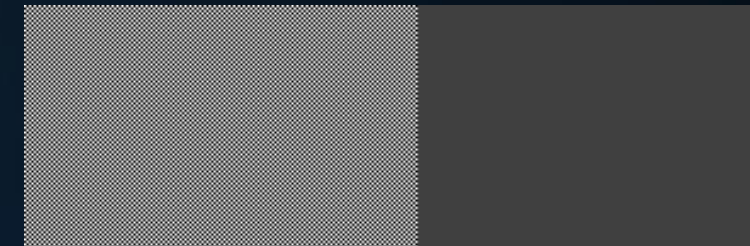
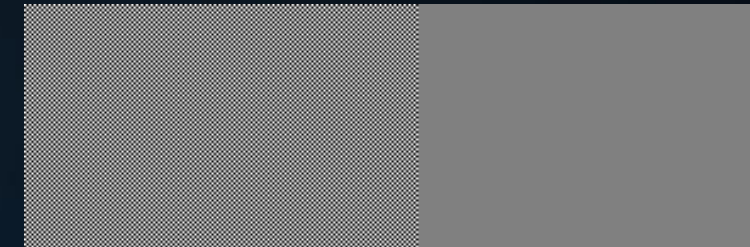
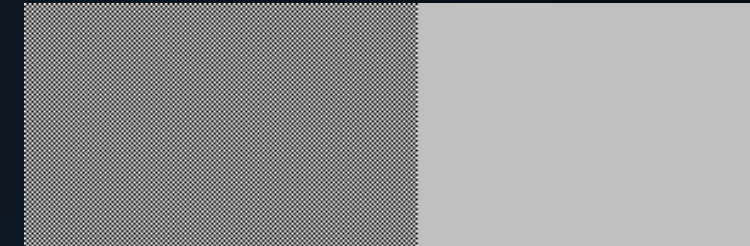
Monitors respond in a non-linear fashion to input:

Displayed intensity  $I = I_{max} a^\gamma$

Example for  $\gamma=2$ :  $a = \left\{0, \frac{1}{2}, 1\right\} \rightarrow I = \left\{0, \frac{1}{4}, 1\right\}$

Let's see what  $\gamma$  is on the beamer. ☺

*On most monitors,  $\gamma \approx 2$ .*





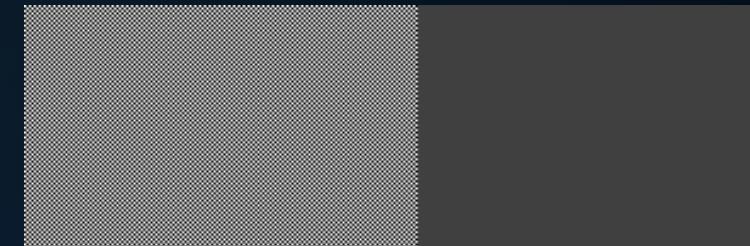
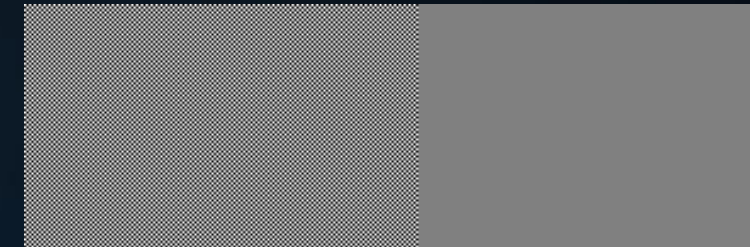
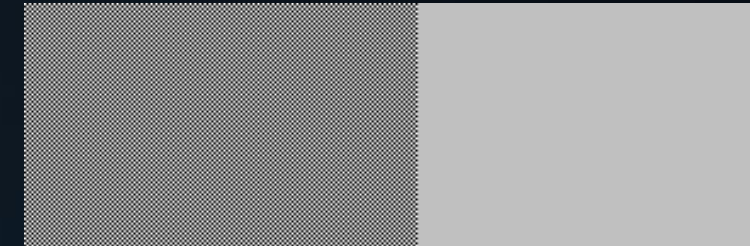
# Gamma Correction

How to deal with  $\gamma \approx 2$

First of all: we will want to do our rendering calculations in a linear fashion.

Assuming that we did this, we will want an intensity of 50% to show up as 50% brightness.

Knowing that  $I = a^\gamma$ ,  
 we adjust the input:  $a' = a^{\frac{1}{\gamma}}$  (for  $\gamma=2$ ,  $a' = \sqrt{a}$ ),  
 so that  $I = a'^\gamma = (a^{\frac{1}{\gamma}})^\gamma = a$ .



# Gamma Correction

How to deal with  $\gamma \approx 2$

Apart from ‘gamma correcting’ our output, we also need to pay attention to our input.

This photo looks as good as it does because it was adjusted for screens with  $\gamma \approx 2$ .

In other words: the intensities stored in this image file have been processed so that  $a^\gamma$  yields the intended intensity; i.e. linear values  $a$  have

been adjusted:  $a' = a^{\frac{1}{\gamma}}$ .

We restore the linear values for the image as follows:

$$a = a'^\gamma$$





# Gamma Correction

## Linear workflow

To ensure correct (linear) operations:

1. Input data  $a'$  is linearized:  $a = a'^\gamma$
2. All calculations assume linear data
3. Final result is gamma corrected:  $a' = a^{\frac{1}{\gamma}}$
4. The monitor applies a non-linear scale to obtain the final linear result  $a$ .

Interesting fact: modern monitors have no problem at all displaying linear intensity curves: they are forced to use a non-linear curve because of legacy...



# Today's Agenda:

- The Postprocessing Pipeline
  - Vignetting, Chromatic Aberration
  - Film Grain
  - HDR effects
  - Color Grading
  - Depth of Field
- Screen Space Algorithms
  - Ambient Occlusion
  - Screen Space Reflections
  - Anti-aliasing





# Depth of Field

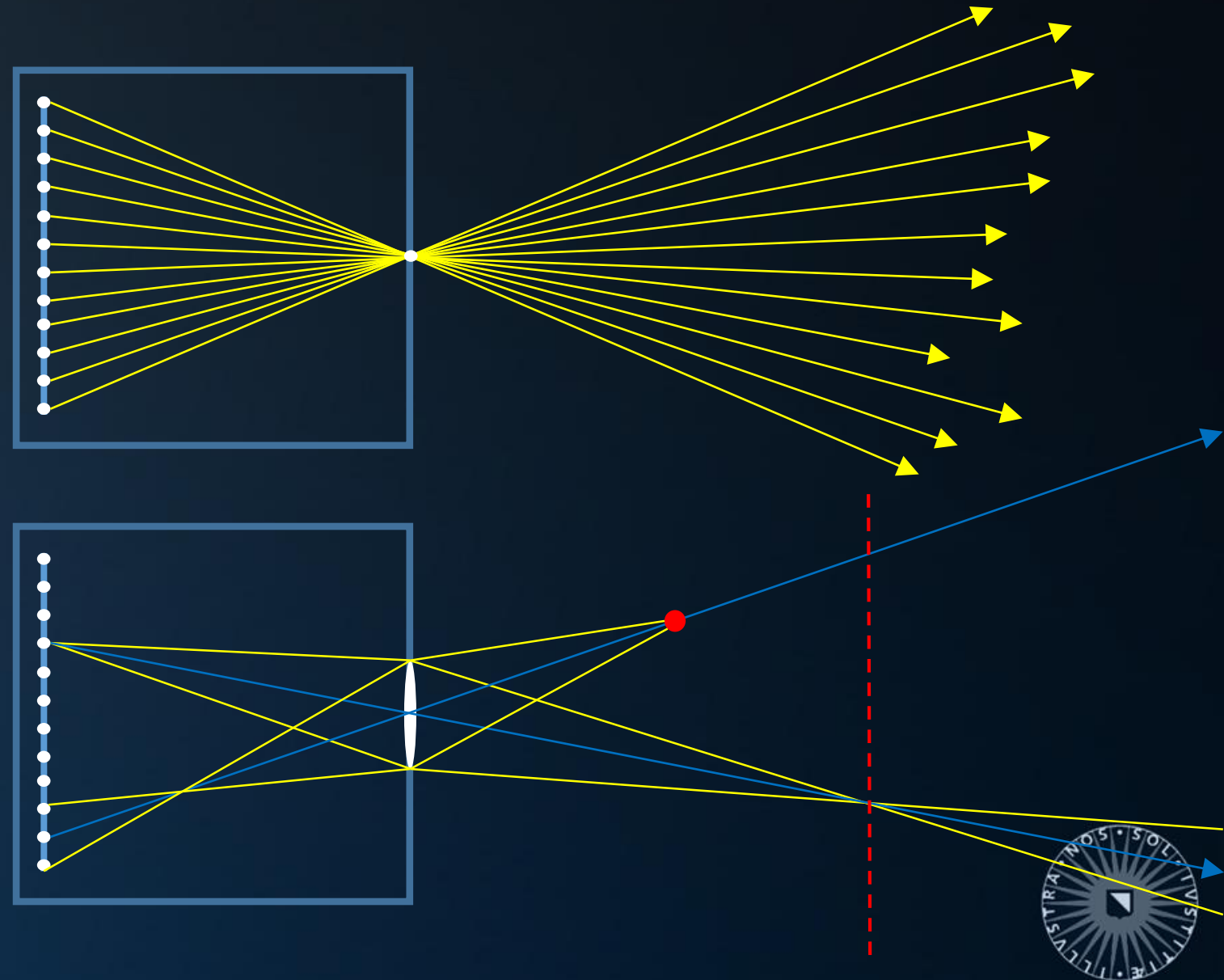
A pinhole camera maps incoming directions to pixels.

Pinhole: aperture size = 0

For aperture sizes  $> 0$ , the lens has a focal distance.

Objects not precisely at that distance cause incoming light to be spread out over an area, rather than a point on the film.

This area is called the ‘circle of confusion’.



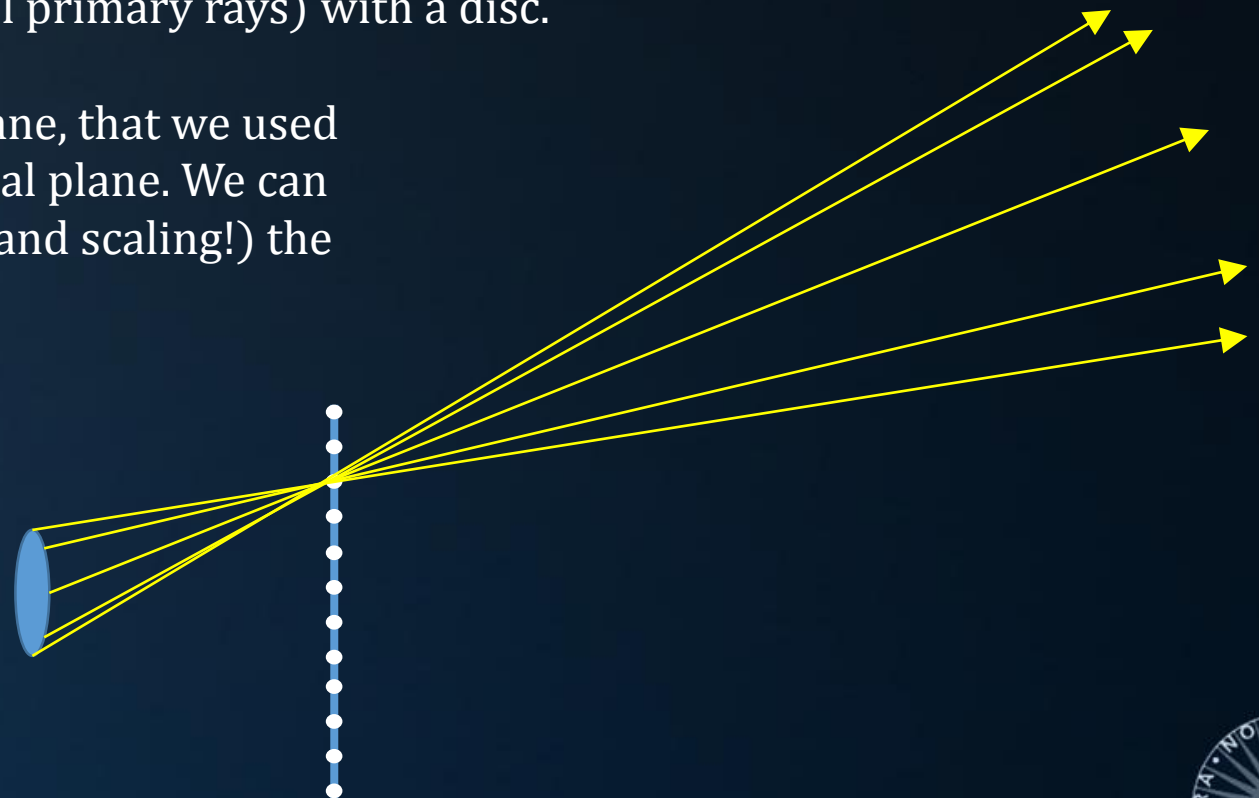
# Depth of Field

## Depth of Field in a Ray Tracer

To model depth of field in a ray tracer, we exchange the pinhole camera (i.e., a single origin for all primary rays) with a disc.

Notice that the virtual screen plane, that we used to aim our rays at, is now the focal plane. We can shift the focal plane by moving (and scaling!) the virtual plane.

We generate primary rays, using Monte-Carlo, on the ‘lens’.





# Depth of Field

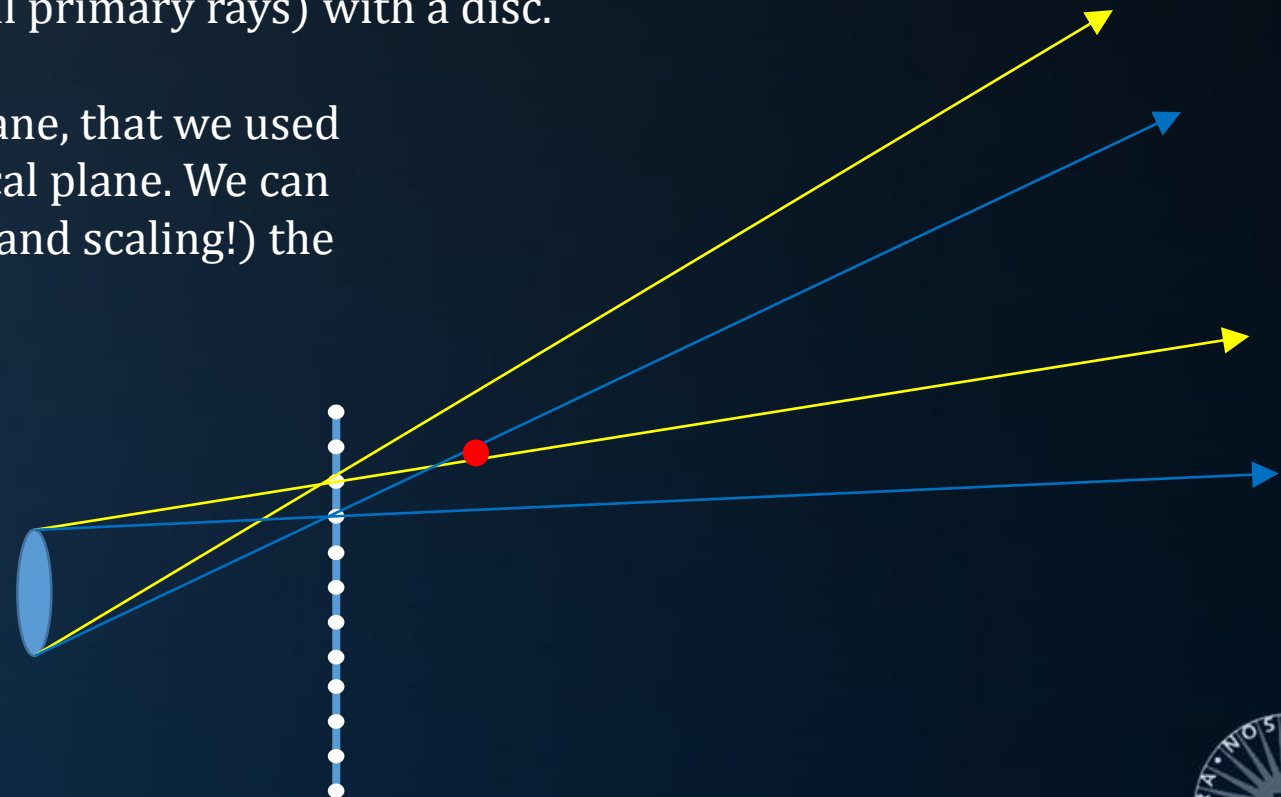
## Depth of Field in a Ray Tracer

To model depth of field in a ray tracer, we exchange the pinhole camera (i.e., a single origin for all primary rays) with a disc.

Notice that the virtual screen plane, that we used to aim our rays at, is now the focal plane. We can shift the focal plane by moving (and scaling!) the virtual plane.

We generate primary rays, using Monte-Carlo, on the ‘lens’.

The red dot is now detected by two pixels.



# Depth of Field

## Depth of Field in a Rasterizer

Depth of field in a rasterizer can be achieved in several ways:

1. Render the scene from several view points, and average the results;
2. Split the scene in layers, render layers separately, apply an appropriate blur to each layer and merge the results;
3. Replace each pixel by a disc sprite, and draw this sprite with a size matching the circle of confusion;
4. Filter the ‘in-focus’ image to several buffers, and blur each buffer with a different kernel size. Then, for each pixel select the appropriate blurred buffer.
5. As a variant on 4, just blend between a single blurred buffer and the original one.

Note that in all cases (except 1), the input is still an image generated by a pinhole camera.





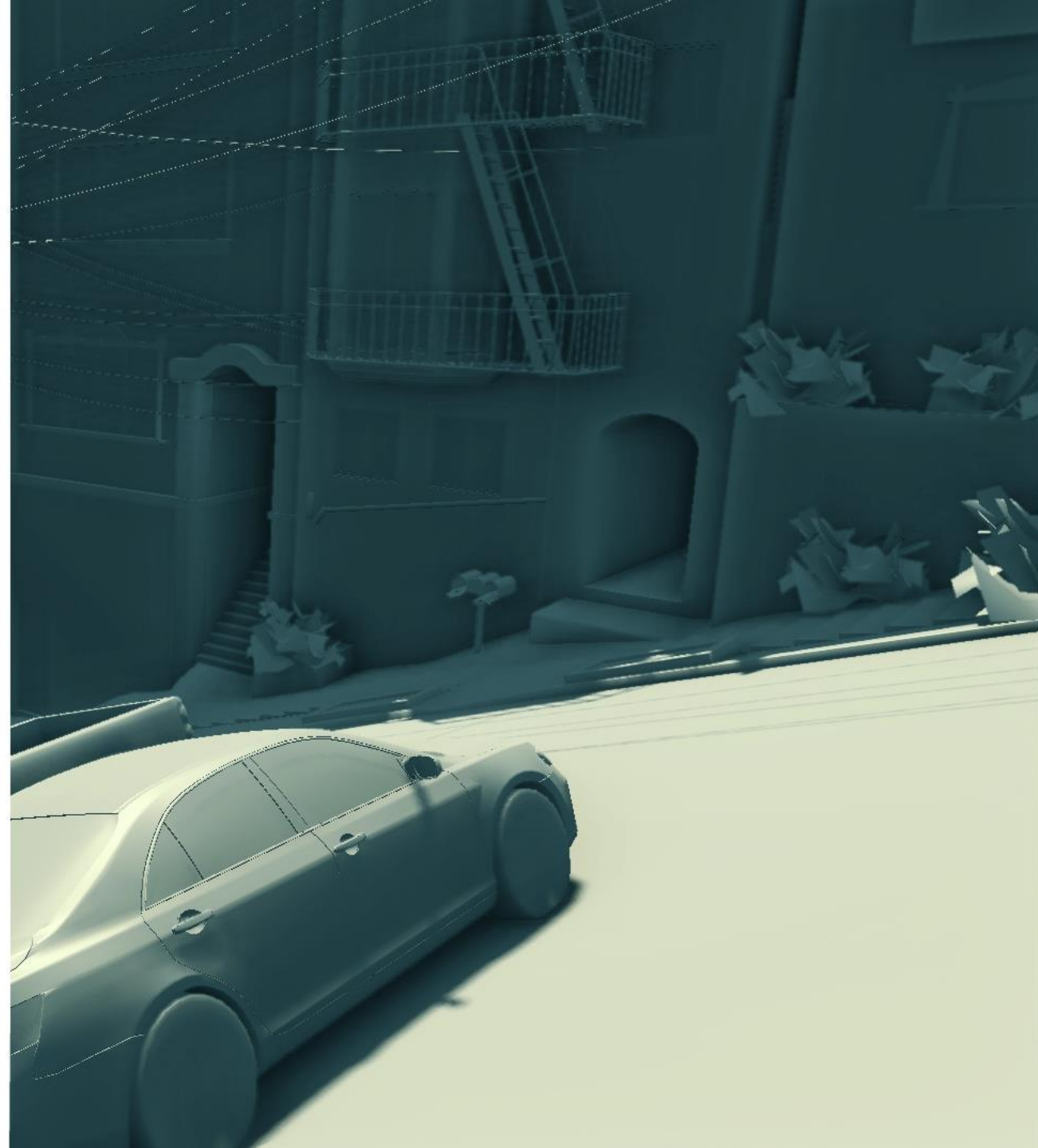


# Today's Agenda:

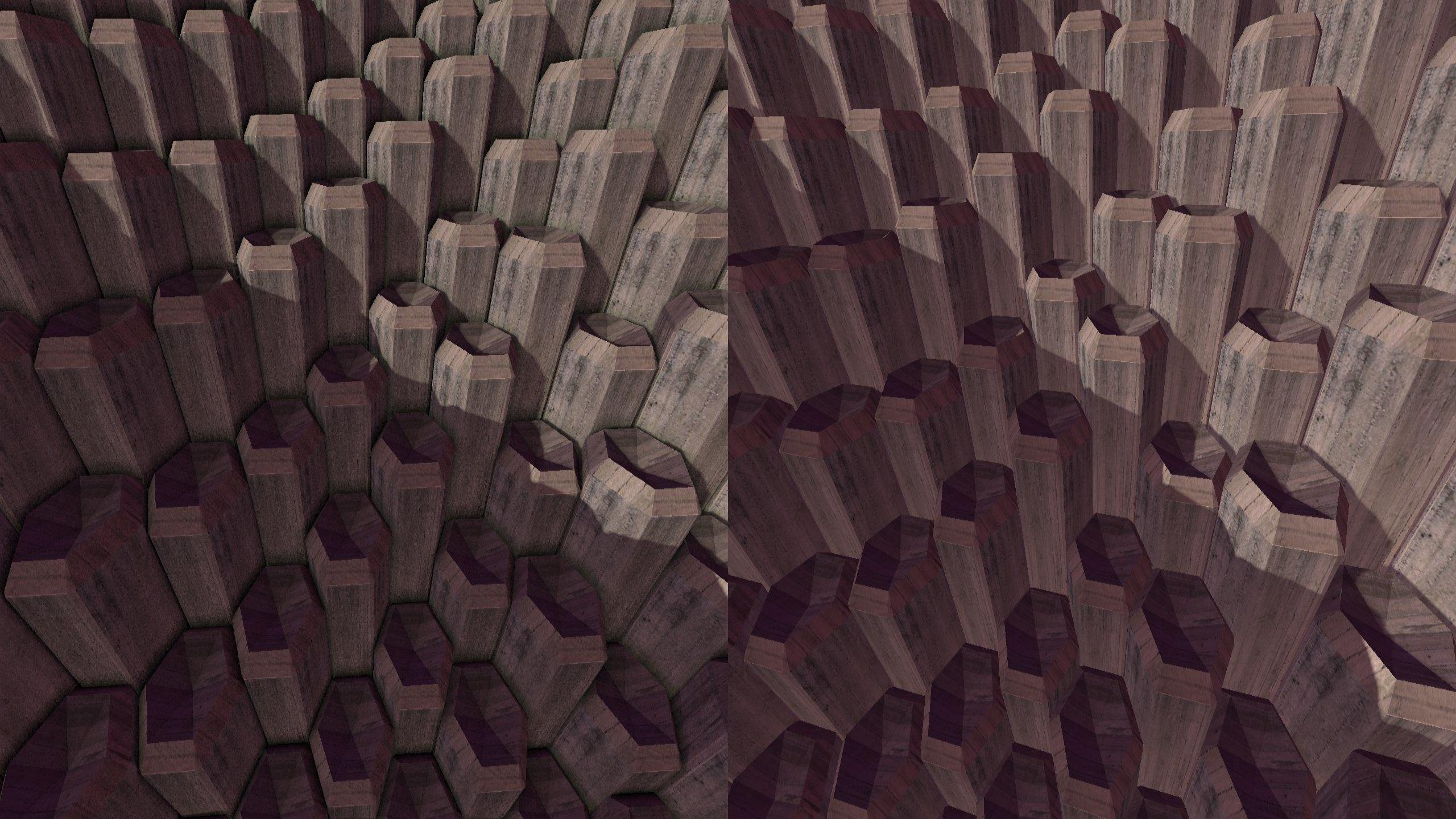
- The Postprocessing Pipeline
  - Vignetting, Chromatic Aberration
  - Film Grain
  - HDR effects
  - Color Grading
  - Depth of Field
- Screen Space Algorithms
  - Ambient Occlusion
  - Screen Space Reflections













324 MHz  
1071 MHz  
790 MB  
179.9 FPS

WAVE 1

TAKE DOWN ALL HOSTILES

HOSTILES REMAINING

5

[SPACE] CLIMB



10M



STICKY NOISEMAKER

3



11 30  
SILENCED



```
GPU1 : 41 °C, 7 %, 95 %, 324 MHz
GPU2 : 68 °C, 95 %, 95 %, 1019 MHz
MEM1 : 324 MHz, 838 MB
MEM2 : 3005 MHz, 838 MB
D3011 : 82.6 FPS
```

▲



3



**11:30**  
**SILENCED**

## SILENCED



# Ambient Occlusion

## Concept

Ambient occlusion was designed to be a scale factor for the ambient factor in the Phong shading model.

A city under a skydome:  
assuming uniform illumination  
from the dome, illumination of  
the buildings is proportional to  
the visibility of the skydome.



# Ambient Occlusion

## Concept

This also works for much smaller hemispheres:

We test a fixed size hemisphere for occluders.  
The ambient occlusion factor is then either:

- The portion of the hemisphere surface that is visible from the point;
- Or the average distance we can see before encountering an occluder.





# Ambient Occlusion

## Concept

Ambient occlusion is generally determined using Monte Carlo integration, using a set of rays.

$$AO = \frac{1}{N} \sum_{i=1}^N V_{P, \vec{w}} (\vec{N} \cdot \vec{w})$$

where  $V$  is 1 or 0, depending on the visibility of points on the hemisphere at a fixed distance.

or

$$AO = \frac{1}{N} \sum_{i=1}^N \frac{D_{P, \vec{w}}}{D_{max}} (\vec{N} \cdot \vec{w})$$

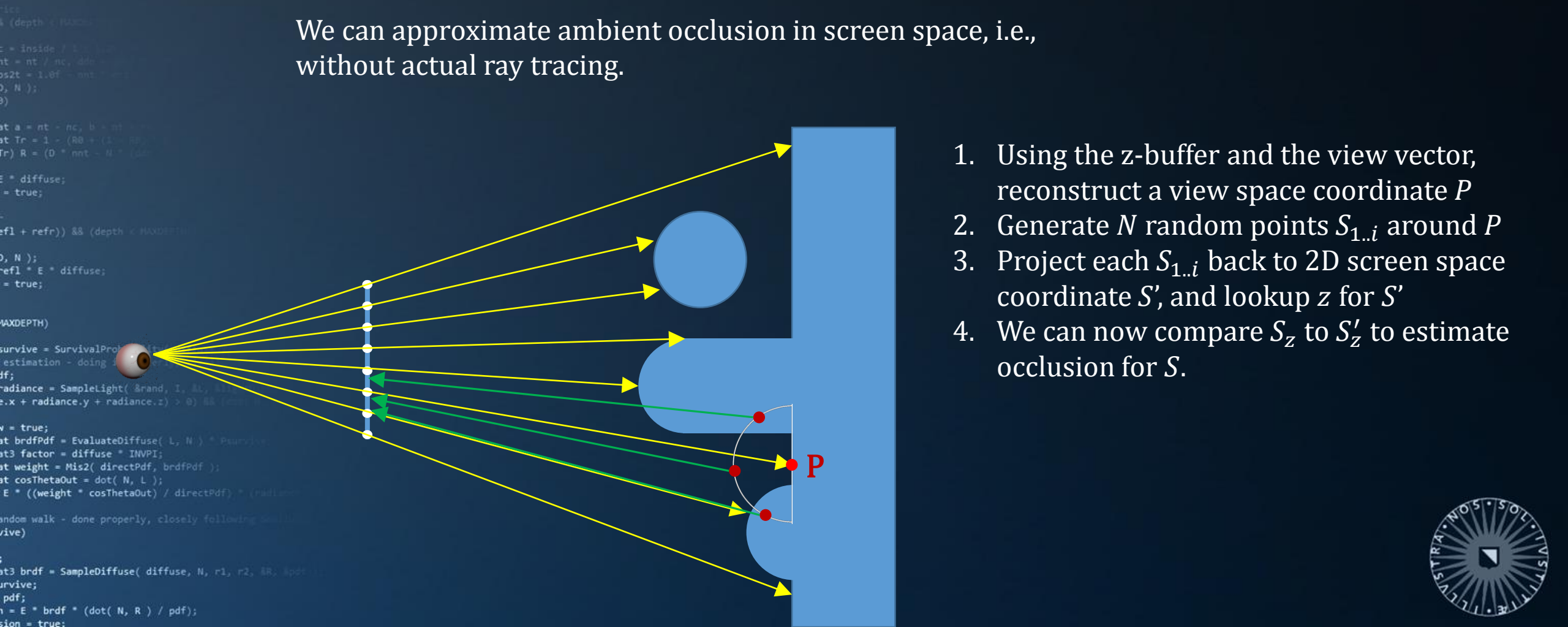
where  $D_{P, \vec{w}}$  is the distance to the first occluder or a point on a hemisphere with radius  $D_{max}$ .



# Ambient Occlusion

## Screen Space Ambient Occlusion

We can approximate ambient occlusion in screen space, i.e., without actual ray tracing.



1. Using the z-buffer and the view vector, reconstruct a view space coordinate  $P$
2. Generate  $N$  random points  $S_{1..i}$  around  $P$
3. Project each  $S_{1..i}$  back to 2D screen space coordinate  $S'$ , and lookup  $z$  for  $S'$
4. We can now compare  $S_z$  to  $S'_z$  to estimate occlusion for  $S$ .







# Ambient Occlusion

## Filtering SSAO

Applying the separable Gaussian blur you implemented already is insufficient for filtering SSAO: we don't want to blur AO values over edges.

We use a *bilateral filter* instead.

Such a filter replaces each value in an image by a weighted average of nearby pixels. Instead of using a fixed weight, the weight is computed on the fly, e.g. based on the view space distance of two points, or the dot between normals for the two pixels.





# Today's Agenda:

- The Postprocessing Pipeline
  - Vignetting, Chromatic Aberration
  - Film Grain
  - HDR effects
  - Color Grading
  - Depth of Field
- Screen Space Algorithms
  - Ambient Occlusion
  - Screen Space Reflections

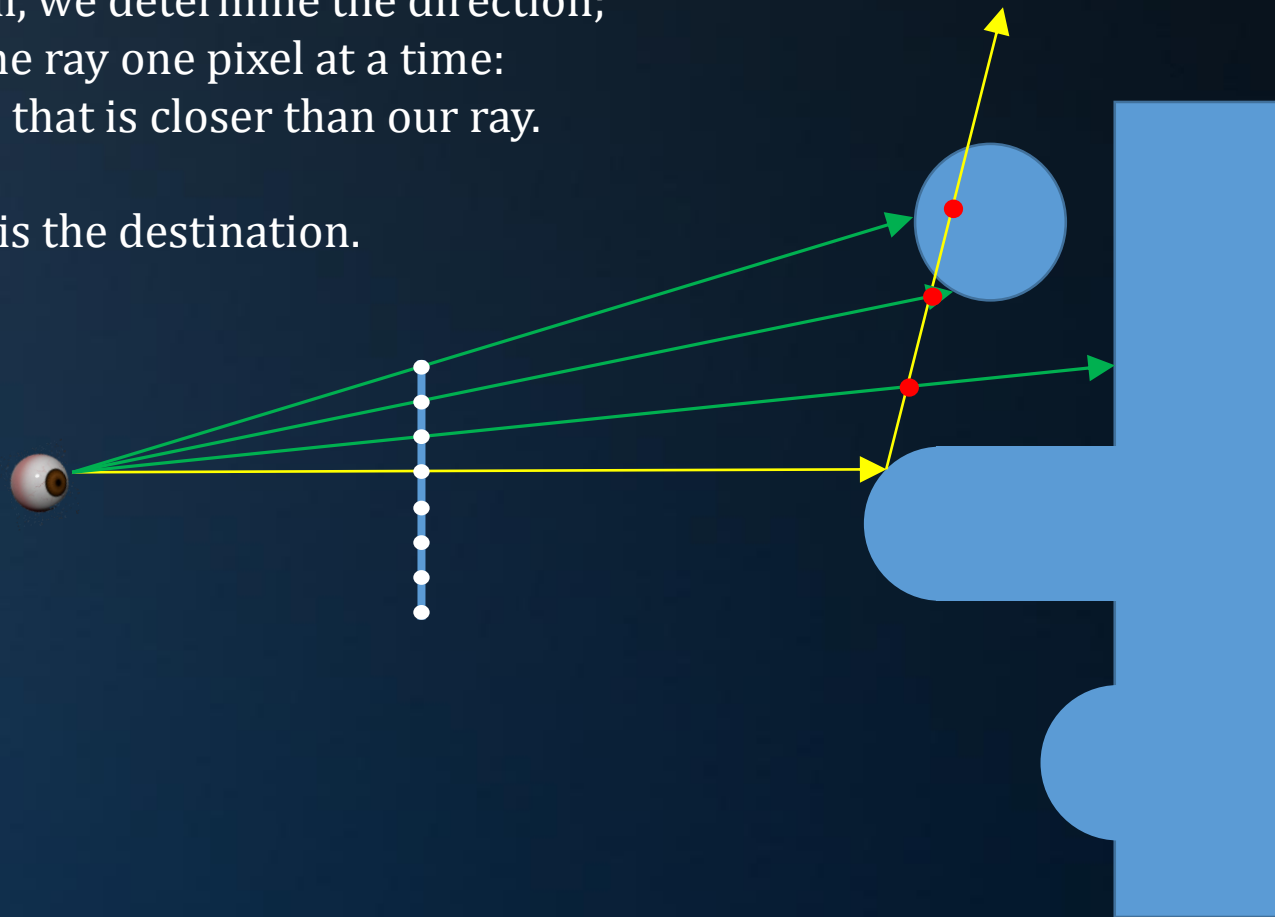


# Reflections

## Screen Space Reflections

1. Based on depth, we determine the origin of the ray;
2. Based on normal, we determine the direction;
3. We step along the ray one pixel at a time:
4. Until we find a z that is closer than our ray.

The previous point is the destination.





# Reflections

## Screen Space Reflections

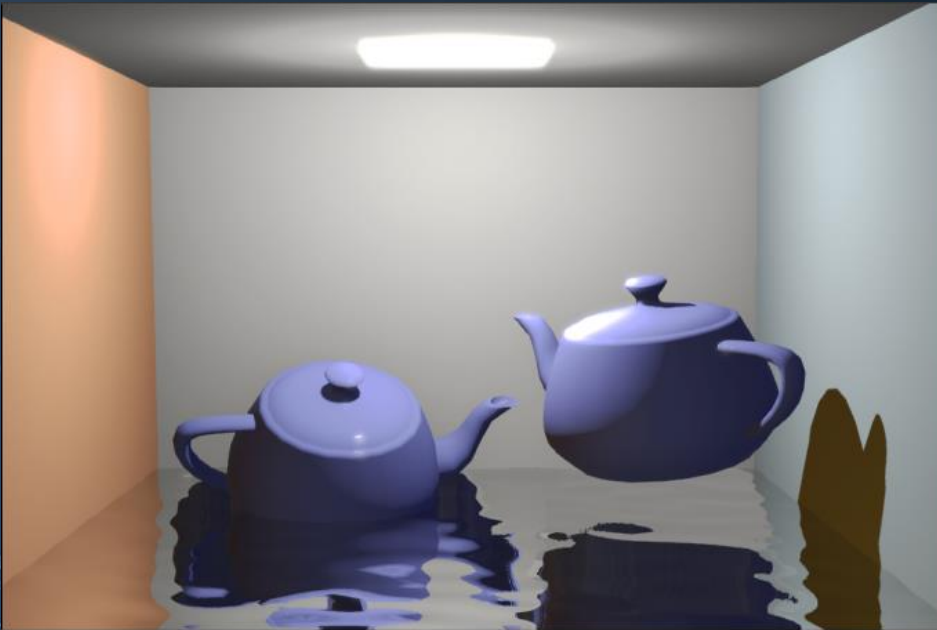


From: <http://www.code80.com/blog/2015/03/11/screen-space-reflections-in-unity-5>



# Reflections

## Screen Space Reflections



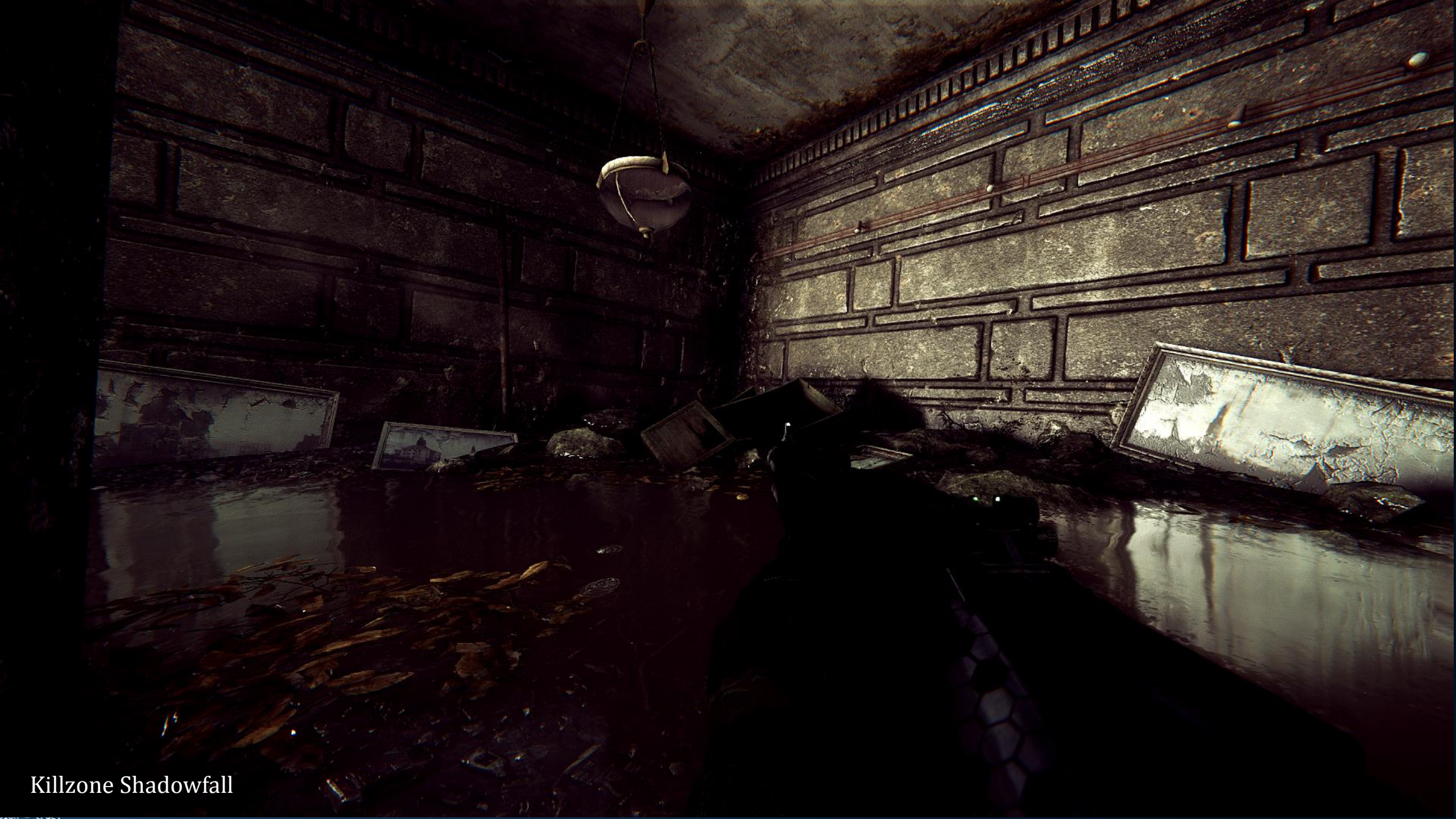
“Efficient GPU Screen-Space Ray Tracing”, McGuire & Mara, 2014













# Today's Agenda:

- The Postprocessing Pipeline
  - Vignetting, Chromatic Aberration
  - Film Grain
  - HDR effects
  - Color Grading
  - Depth of Field
- Screen Space Algorithms
  - Ambient Occlusion
  - Screen Space Reflections



# Famous Last Words

## Post Processing Pipeline

In: rendered image, linear color space

- Ambient occlusion
- Screen space reflections
- Tone mapping
- HDR bloom / glare
- Depth of field
- Film grain / vignetting / chromatic aberration
- Color grading
- Gamma correction

Out: post-processed image, gamma corrected

```

 // ...
 // (depth < MAXDEPTH)
 // ...
 // inside / 1.0f;
 // nt = nt / nc; add = add + nt;
 // pos2t = 1.0f - nnt; // ...
 // D, N);
 // ...
 // ...
 // at a = nt - nc; b = nt - nc;
 // at Tr = 1 - (R0 + (1 - R0) * ...
 // Tr) R = (D * nnt - N * (2 * ...
 // ...
 // E * diffuse;
 // ...
 // ...
 // refl + refr)) && (depth < MAXDEPTH)
 // ...
 // D, N);
 // refl * E * diffuse;
 // ...
 // ...
 // MAXDEPTH)
 // ...
 // survive = SurvivalProbability(diffuse, ...
 // estimation - doing it properly, closely ...
 // if;
 // radiance = SampleLight(&rand, I, &t, &light ...
 // e.x + radiance.y + radiance.z) > 0) && (depth < ...
 // ...
 // v = true;
 // at brdfPdf = EvaluateDiffuse(L, N) * Psurvive;
 // at3 factor = diffuse * INVPI;
 // at weight = Mix2(directPdf, brdfPdf);
 // at cosThetaOut = dot(N, L);
 // E * ((weight * cosThetaOut) / directPdf) * (radiance ...
 // ...
 // random walk - done properly, closely following ...
 // survive)
 // ...
 // at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf ...
 // survive;
 // pdf;
 // n = E * brdf * (dot(N, R) / pdf);
 // ...
 // ...
 // ...

```





# Today's Agenda:

- The Postprocessing Pipeline
  - Vignetting, Chromatic Aberration
  - Film Grain
  - HDR effects
  - Color Grading
  - Depth of Field
- Screen Space Algorithms
  - Ambient Occlusion
  - Screen Space Reflections



# INFOGR – Computer Graphics

J. Bikker - April-July 2015 - Lecture 12: “Advanced Shading”

## END of “Advanced Shading”

final lecture: “Grand Recap”

