Ray Tracing for Games

Dr. Jacco Bikker - IGAD/BUAS, Breda, January 29

Welcome!

Agenda:

- Introduction / Goals / Planning
- Template / LH2
- Whitted-style recap
- RT-centric Game Dev

Ray Tracing for Games

Wednesday
13:00 – 17:00

Thursday
09:00 – 14:00

Friday
09:00 – 17:00

Monday
09:00 – 17:00

Tuesday
09:00 – 17:00

Thursday
09:00 – 17:00

Friday
09:00 – 17:00

Ray Tracing for Games

Whitted

BVH

dielectrics, Beer, correctness

gamma, tonemapping

real-time simple Whitted scene

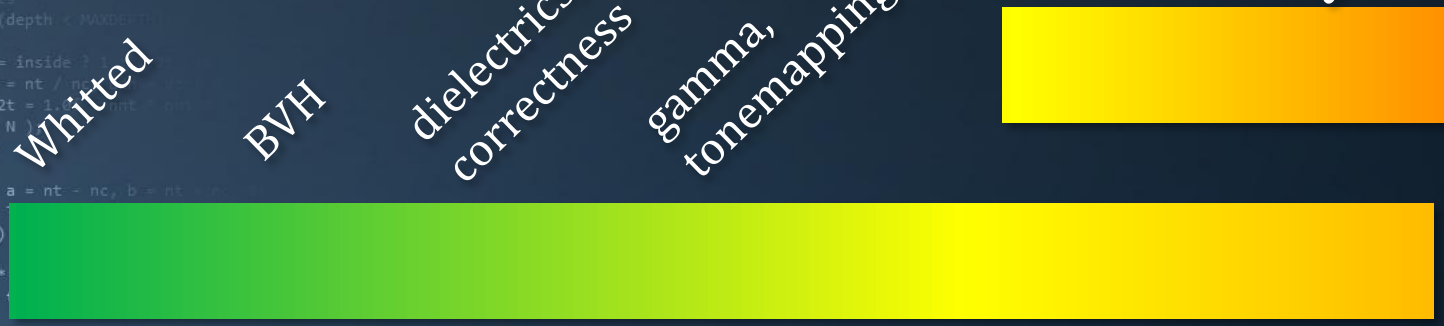
area lights, DOF

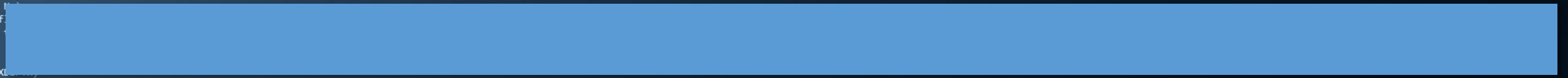
animated BVH

GPGPU

RTX / Optix

real-time GPU path tracing



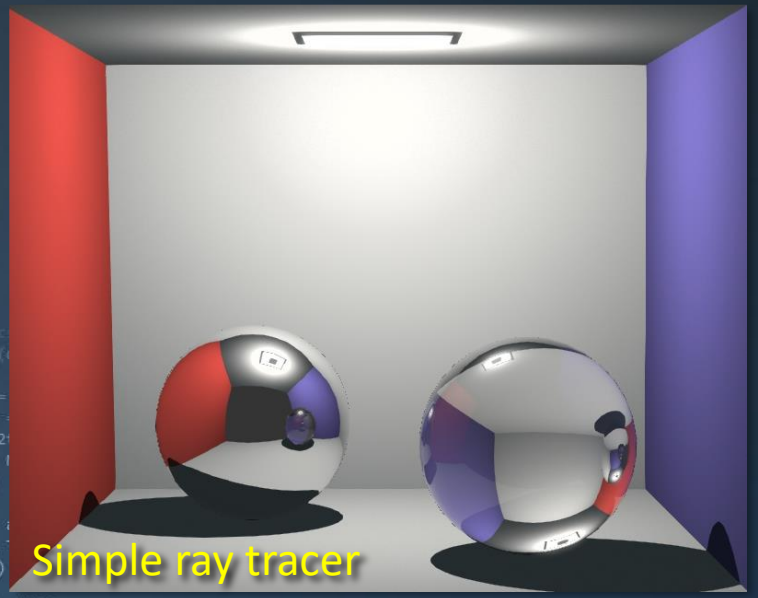


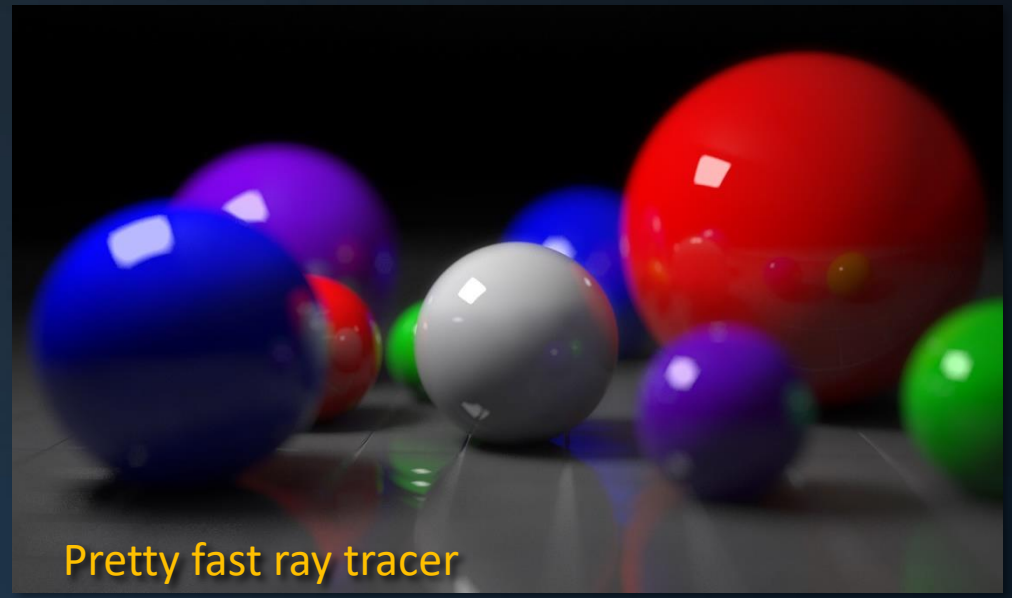
RT game

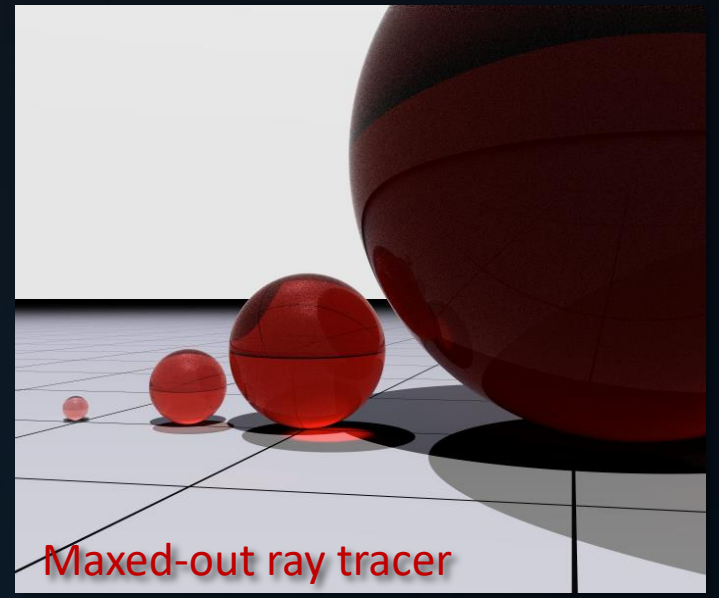
SIMD

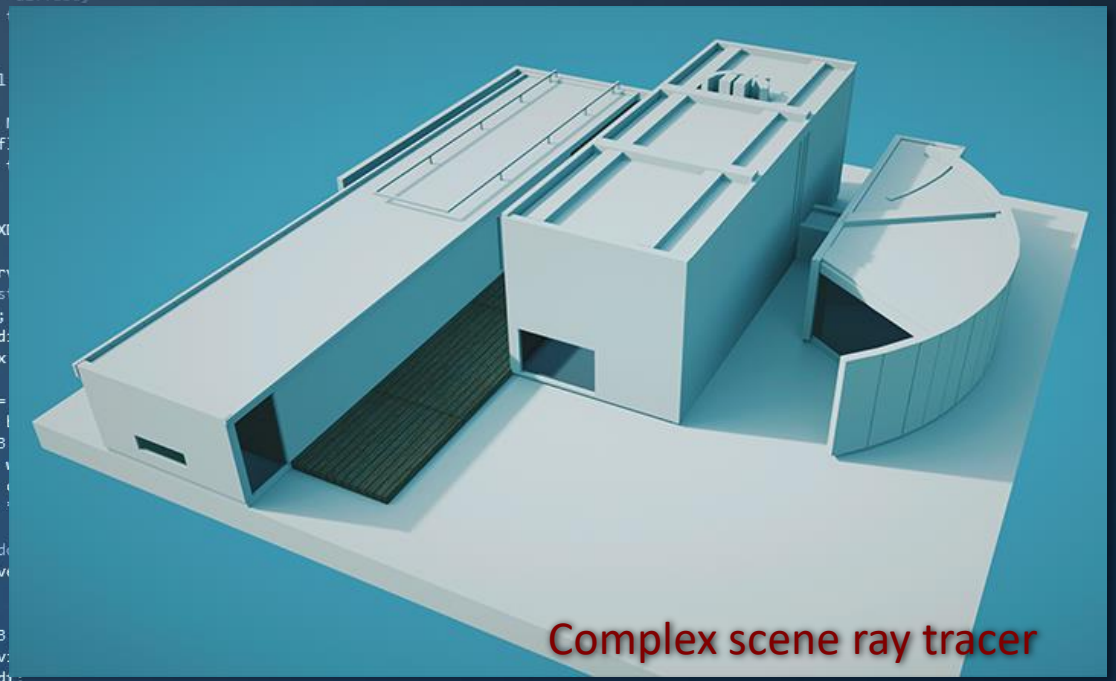
optimization

Ray Tracing for Games

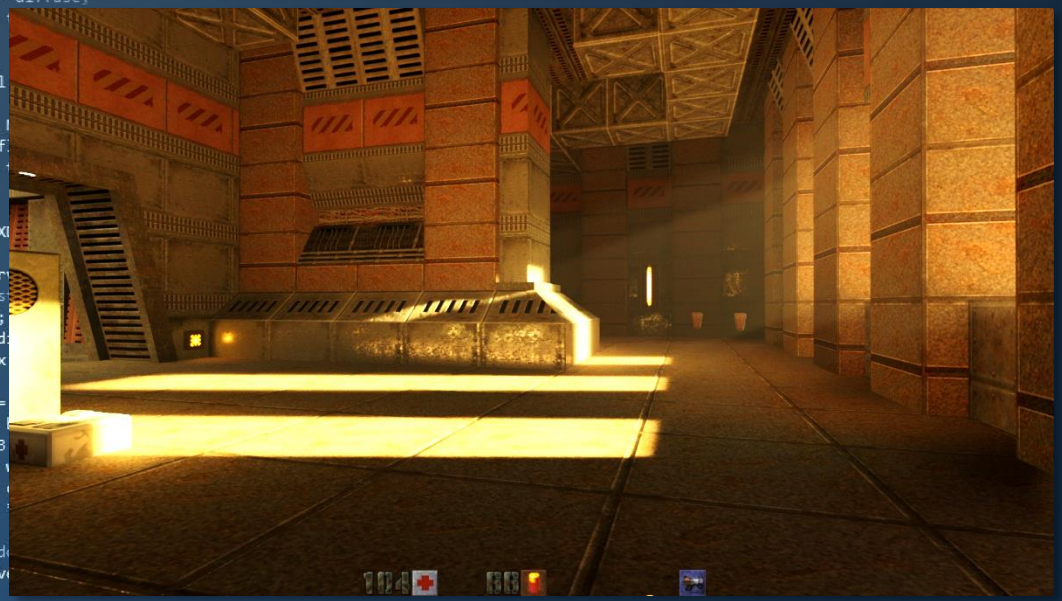


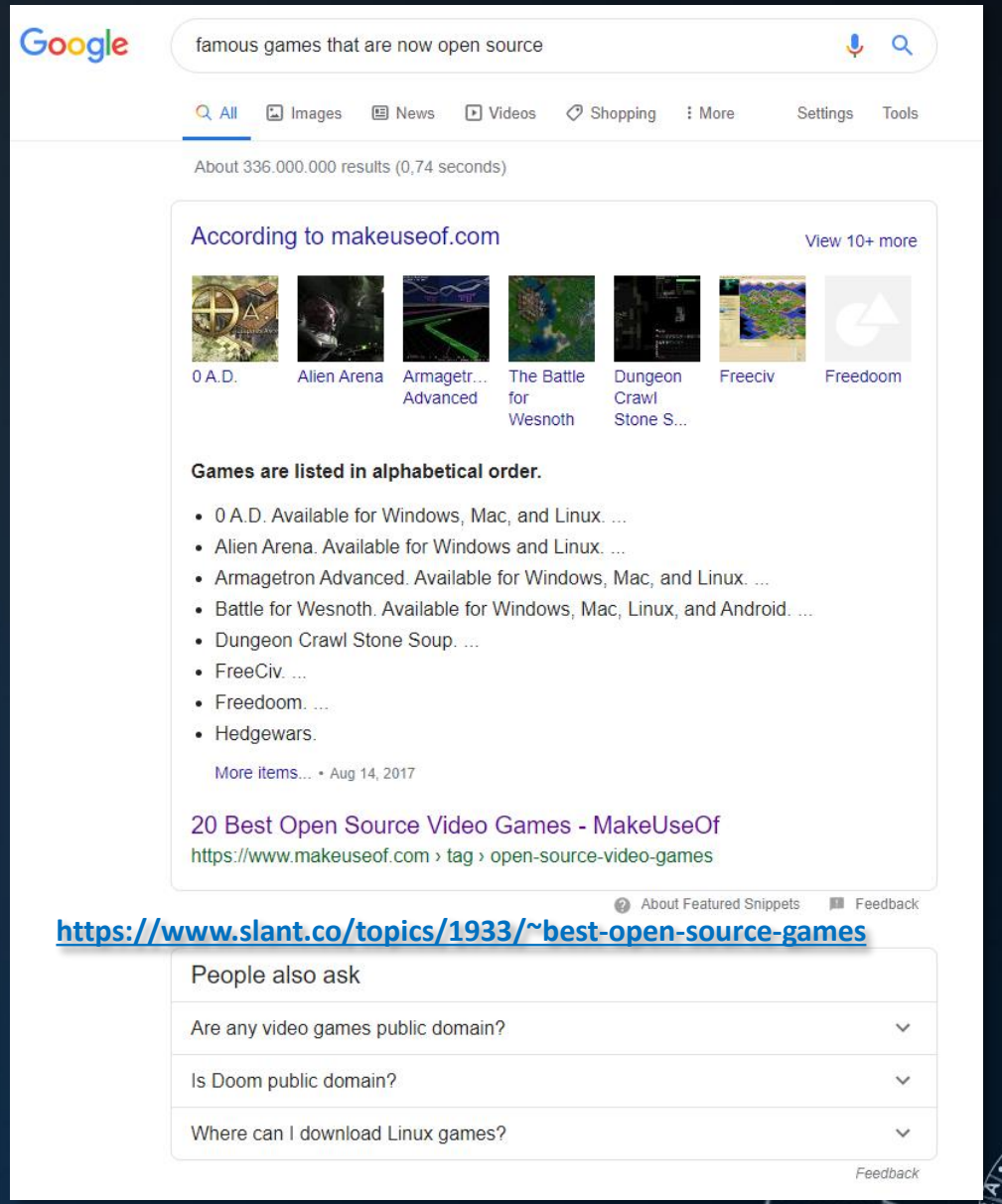






Ray Tracing for Games






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

Ray Tracing for Games

Ray Tracing for Games

Wednesday
13:00 – 17:00

course intro
LH2
template
Whitted
refactoring
RT-centric games

LAB 1

Thursday
09:00 – 14:00

advanced Whitted
audio, AI & physics
faster Whitted
Heaven7

LAB 2

work @ home

End result day 2:

A solid Whitted-style ray tracer, as a basis for subsequent work.

Friday
09:00 – 17:00

optimization
profiling, rules of engagement
threading

LAB 3

SIMD
applied SIMD
SIMD triangle
SIMD AABB

LAB 4

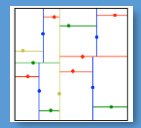
End result day 3:

A 5x faster tracer.

GLOBAL GAME JAM

Monday
09:00 – 17:00

acceleration
grid, BVH, kD-tree
SAH
binning



LAB 5

refitting
top-level BVH
threaded building

LAB 6

End result day 4:

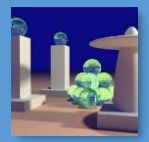
A real-time tracer.

Tuesday
09:00 – 17:00

Monte-Carlo
Cook-style
glossy, AA
area lights, DOF

LAB 7

path tracing



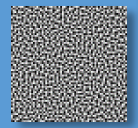
LAB 8

End result day 5:

Cook or Kajiya.

Thursday
09:00 – 17:00

random numbers
stratification
blue noise



LAB 9

importance
sampling
next event
estimation

LAB 10

End result day 6:

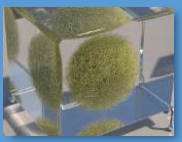
Efficiency.

Friday
09:00 – 17:00

future work

LAB 11

path guiding



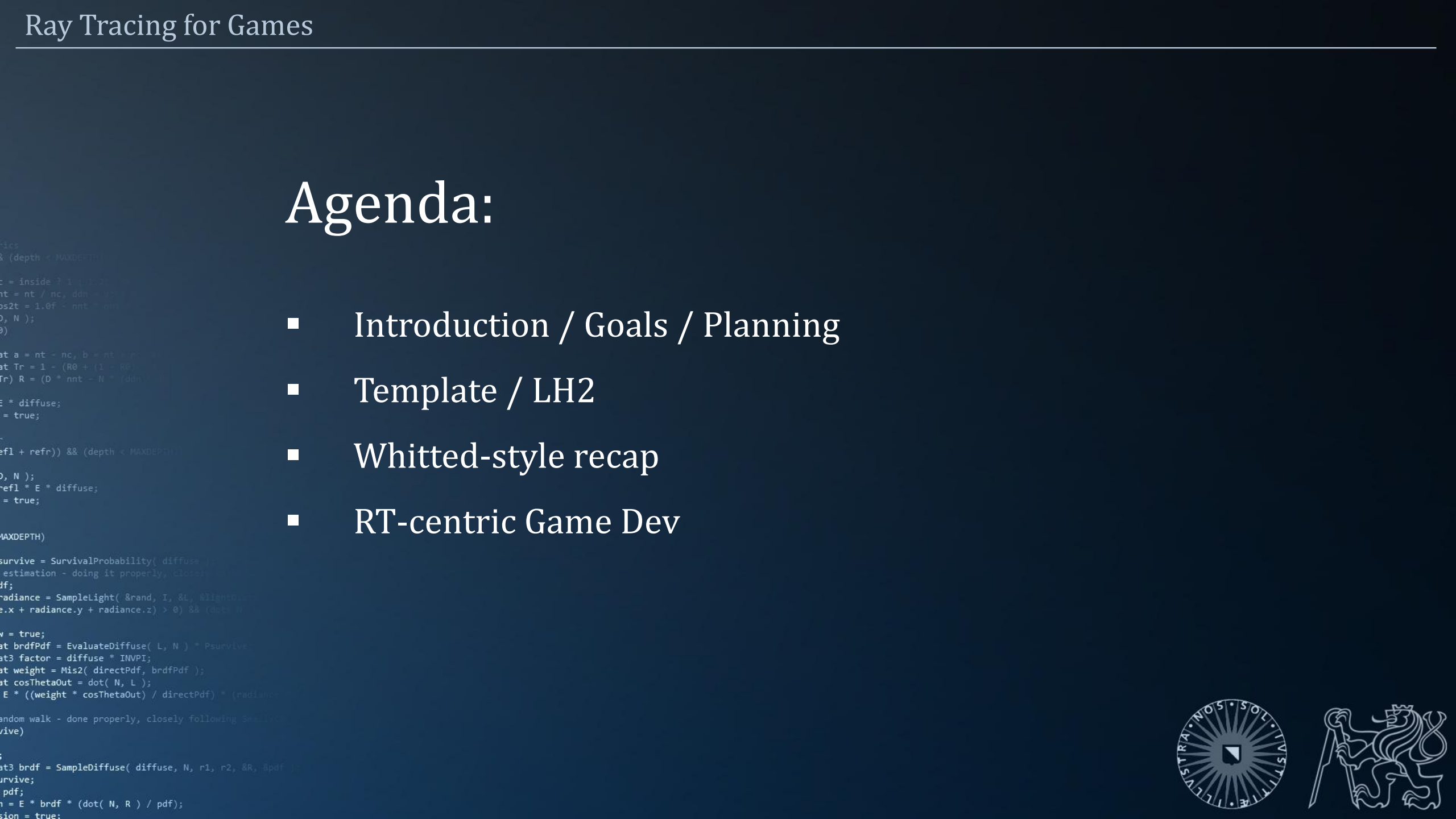
LAB 10

End result day 6:

Great product.

Agenda:

- Introduction / Goals / Planning
- Template / LH2
- Whitted-style recap
- RT-centric Game Dev

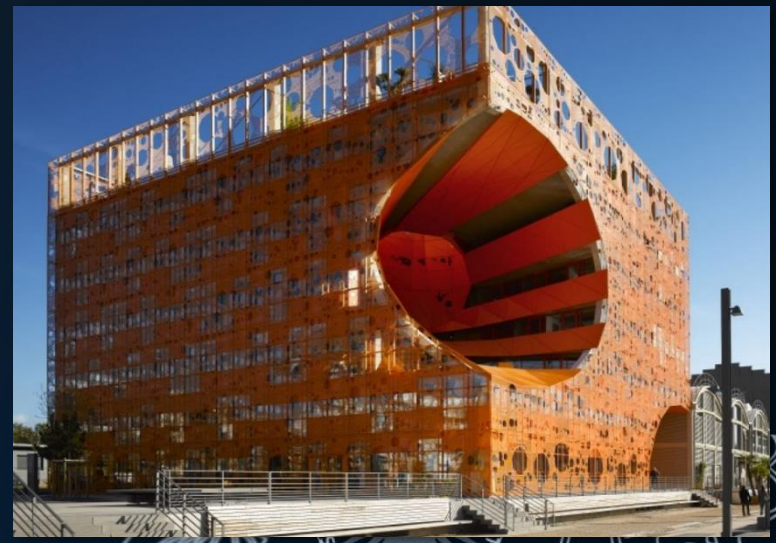


Ray Tracing for Games

1. No Foundation

2. Template

3. Lighthouse 2



Lighthouse 2

Application

- Opens an OpenGL window
- Prepares the scene
- Handles user input
- Calls ‘Render’

(see provided samples)

RenderSystem

- Loads 3D scene data
- Owns & maintains the scene
- Handles animation
- Synchronizes scene data with a ‘Core’
- Invokes a ‘Core’

API overview in:

API ➔ render_api.h

You should not have to change anything in the RenderSystem.

Core

- Receives raw triangle data from RenderSystem
- Performs actual rendering

Several are available:

- Optix7, OptixPrime & Vulkan ray tracing cores;
- A software rasterizer.

Lighthouse 2

Application
- Opens an OpenGL window
- Prepares the scene
- Handles user input
- Calls 'Render'

(several samples provided)

RenderSystem
- Loads 3D scene data
- Owns & maintains the scene
- Handles animation
- Synchronizes scene data with a 'Core'
- Invokes a 'Core'

API overview in:
API ➔ render_api.h

You should not have to change anything in the RenderSystem.

Core
- Receives raw triangle data from RenderSystem
- Performs actual rendering

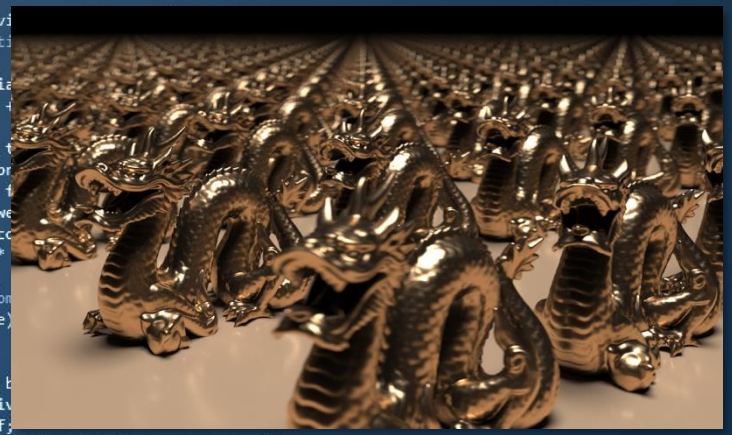
Several are available:
- Optix7, OptixPrime & Vulkan ray tracing cores;
- A software rasterizer.

Lighthouse 2

The Lighthouse 2 system has been designed to facilitate the development of rendering software based on ray tracing.

The RenderSystem takes care of the boring things, such as model loading and scene management. You get to concentrate on the actual ray tracing.

The current system needs a high-end NVIDIA device for pretty images. Without a recent GPU, only a software rasterizer fallback is available. This is where you come in.



Lighthouse 2

Note:
The core is expected to copy the data it receives. There is no guarantee at all that the RenderSystem leaves the provided data intact after SetGeometry returns.

RenderCore

A core implements the low-level rendering functionality.

To build your own: start by copying an existing one.
(project folder ➔ docs ➔ cloning a core.docx)

A core receives raw geometry from the RenderSystem:

```
void RenderCore::SetGeometry(
        const int meshIdx,
        const float4* vertexData,
        const int vertexCount,
        const int triangleCount,
        const CoreTri* triangles,
        const uint* alphaFlags
)
{
}
```

float4: x, y, z, padding

also contain vertices. raw verts are for intersection; tris are for shading.

RenderSystem makes CoreTris from HostTris, CoreMaterials form HostMaterials and so on.

Lighthouse 2

RenderCore

A core implements the low-level rendering functionality.

A core receives instances from the RenderSystem:

```
void RenderCore::SetInstance(
        const int instanceIdx,
        const int meshIdx,
        const mat4& matrix
)
{
}
```

Note:

- A mesh that is not instanced is not visible.
- SetInstance always receives a flattened list of instances; there is no hierarchy.

Lighthouse 2

RenderCore

A core implements the low-level rendering functionality.

A core receives a Render command from the RenderSystem:

```
void RenderCore::Render(
        const ViewPyramid& view,
        const Convergence converge,
        const bool async
)
{
}
```

Note:

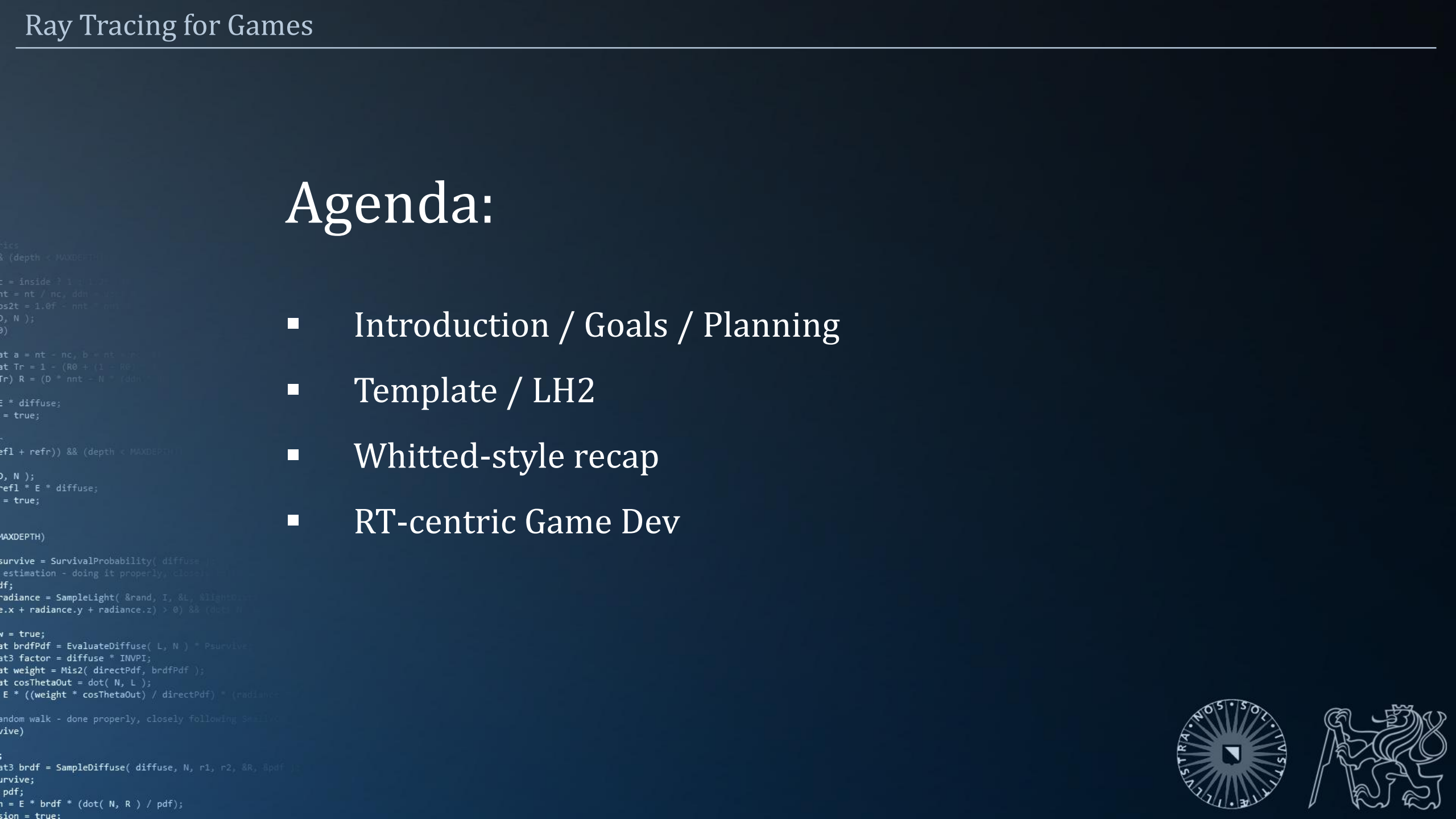
- Converge will be true when the camera is stationary. This will be useful later for path tracing.
- Async can be ignored for now.

Lighthouse 2

```
void Init();
CoreStats GetCoreStats();
void SetProbePos( int2 pos );
void SetTarget( GLTexture* target, uint spp );
void Setting( char* name, float value );
void Render( ViewPyramid& view, Convergence converge, float brightness, float contrast );
void Shutdown();
void SetTextures( CoreTexDesc* tex, int textureCount );
void SetMaterials( CoreMaterial* mat, CoreMaterialEx* matEx, int materialCount );
void SetLights( CoreLightTri* areaLights, int areaLightCount,
        CorePointLight* pointLights, int pointLightCount,
        CoreSpotLight* spotLights, int spotLightCount,
        CoreDirectionalLight* directionalLights, int directionalLightCount );
void SetSkyData( float3* pixels, uint width, uint height );
void SetGeometry( int meshIdx, float4* vertexData, int vertexCount,
        int triangleCount, CoreTri* triangles, uint* alphaFlags = 0 );
void SetInstance( int instanceIdx, int modelIdx, mat4& transform );
void UpdateToplevel();
```

Agenda:

- Introduction / Goals / Planning
- Template / LH2
- Whitted-style recap
- RT-centric Game Dev



Ray Tracing

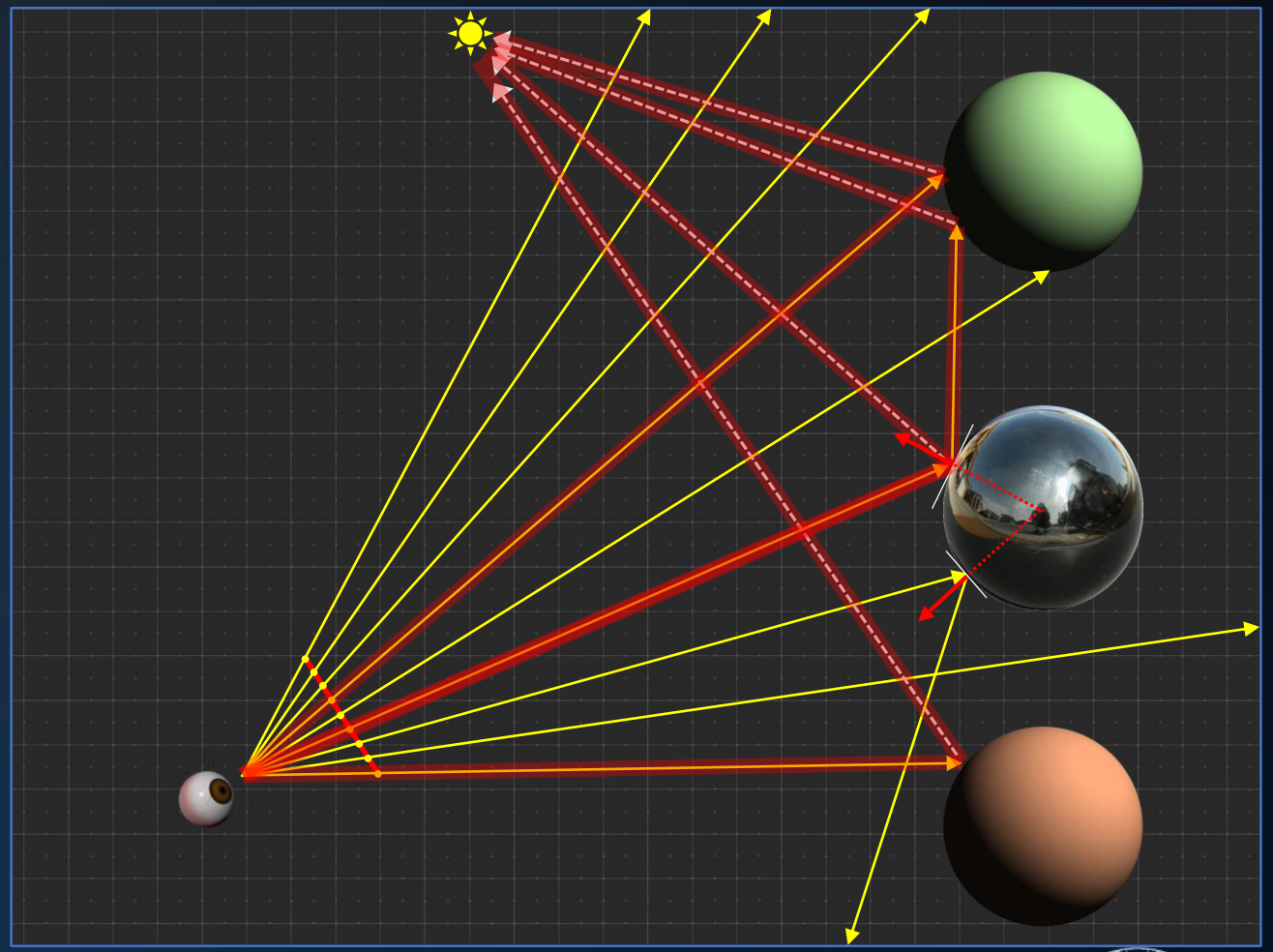
World space

- Geometry
- Eye
- Screen plane
- Screen pixels
- Primary rays
- Intersections
- Point light
- Shadow rays

Light transport

- Extension rays

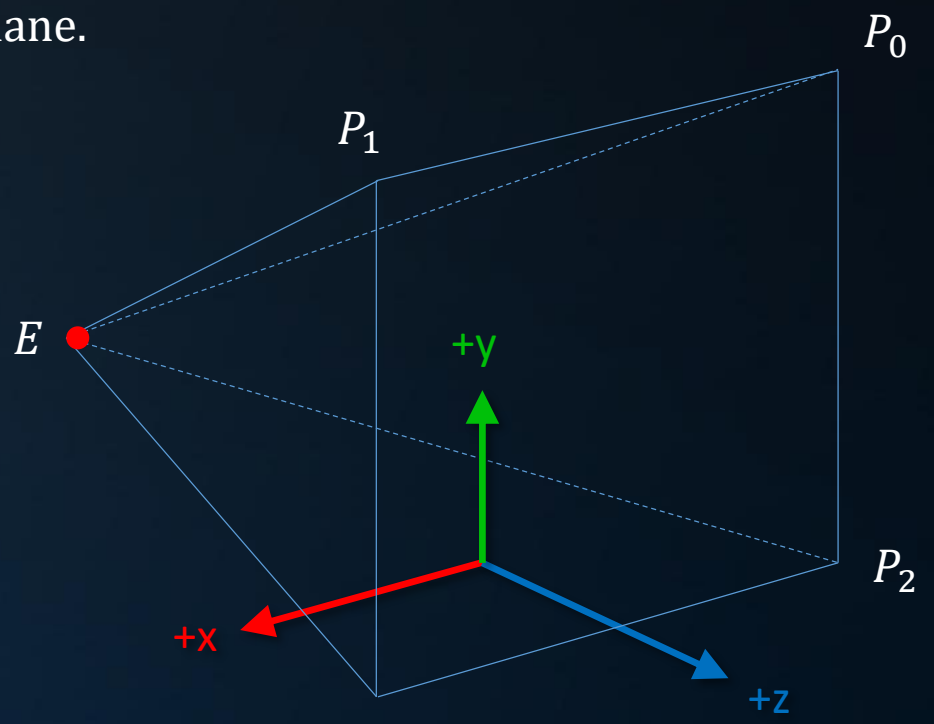
Light transport



Ray setup

A ray is initially shot through a pixel on the screen plane.
The screen plane is defined in *world space*, e.g.:

Camera position: $E = (0,0,0)$
View direction: $\vec{V} = (0,0,1)$
Screen center: $C = E + d\vec{V}$
Screen corners: $P_0 = C + (-1,1,0)$,
$P_1 = C + (1,1,0)$,
$P_2 = C + (-1,-1,0)$



From here:

- Change FOV by altering d;
- Transform camera by multiplying E, P_0, P_1, P_2 with the camera matrix.

Ray setup

do not normalize!

Point on the screen:

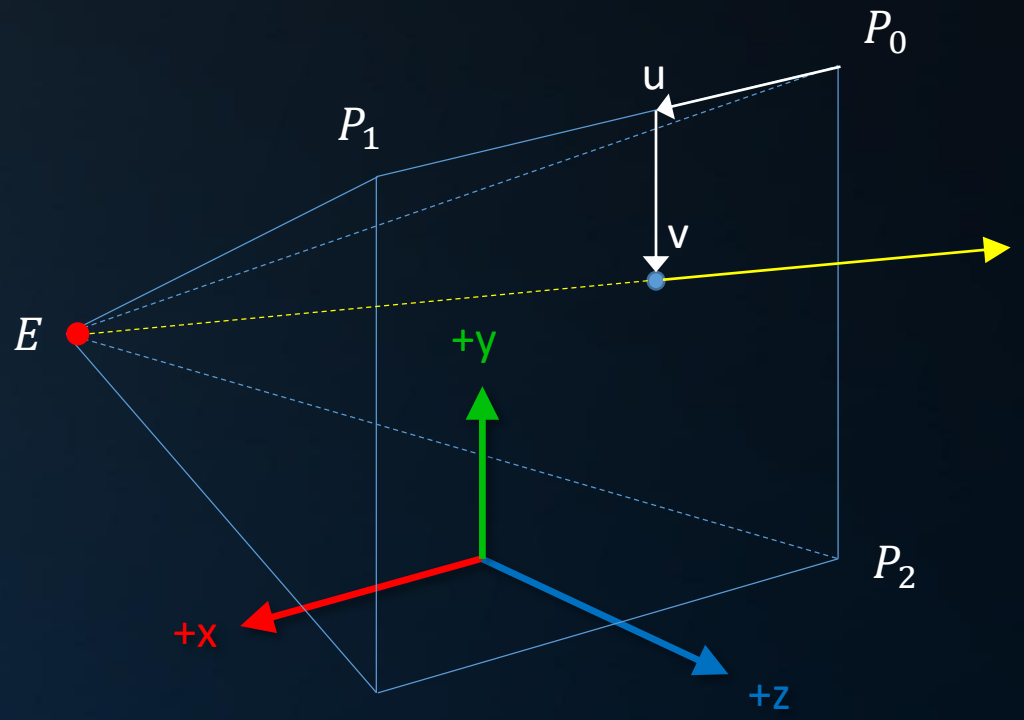
$$P(u,v) = P_0 + u(P_1 - P_0) + v(P_2 - P_0), u, v \in [0,1]$$

Ray direction (normalized):

$$\vec{D} = \frac{P(u,v) - E}{\| P(u,v) - E \|}$$

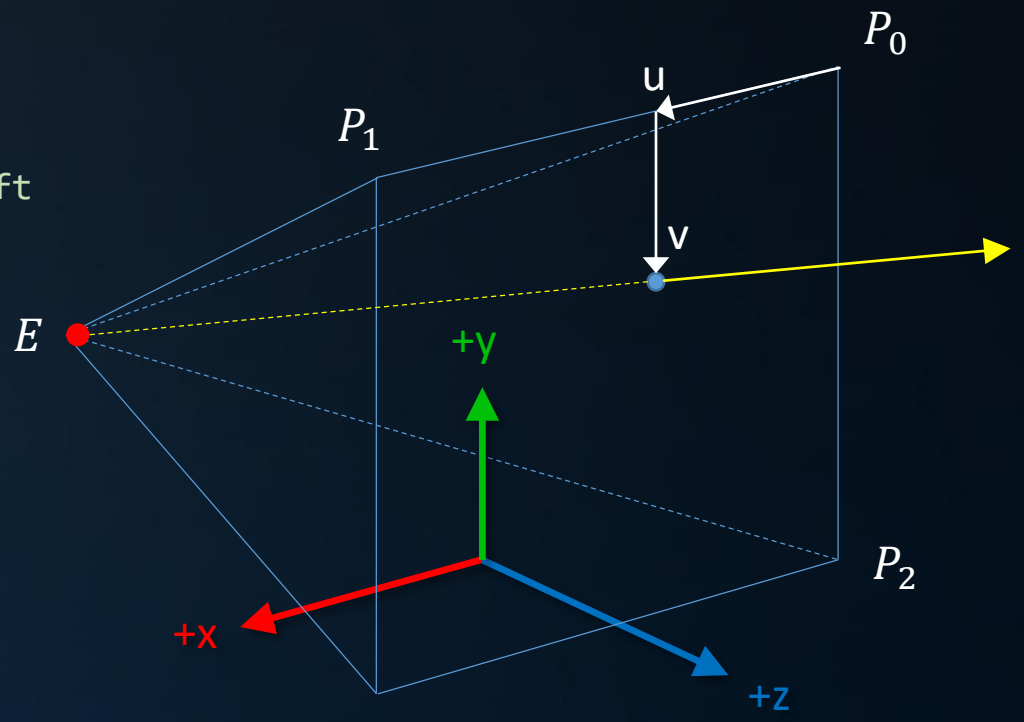
Ray origin:

$$O = E$$



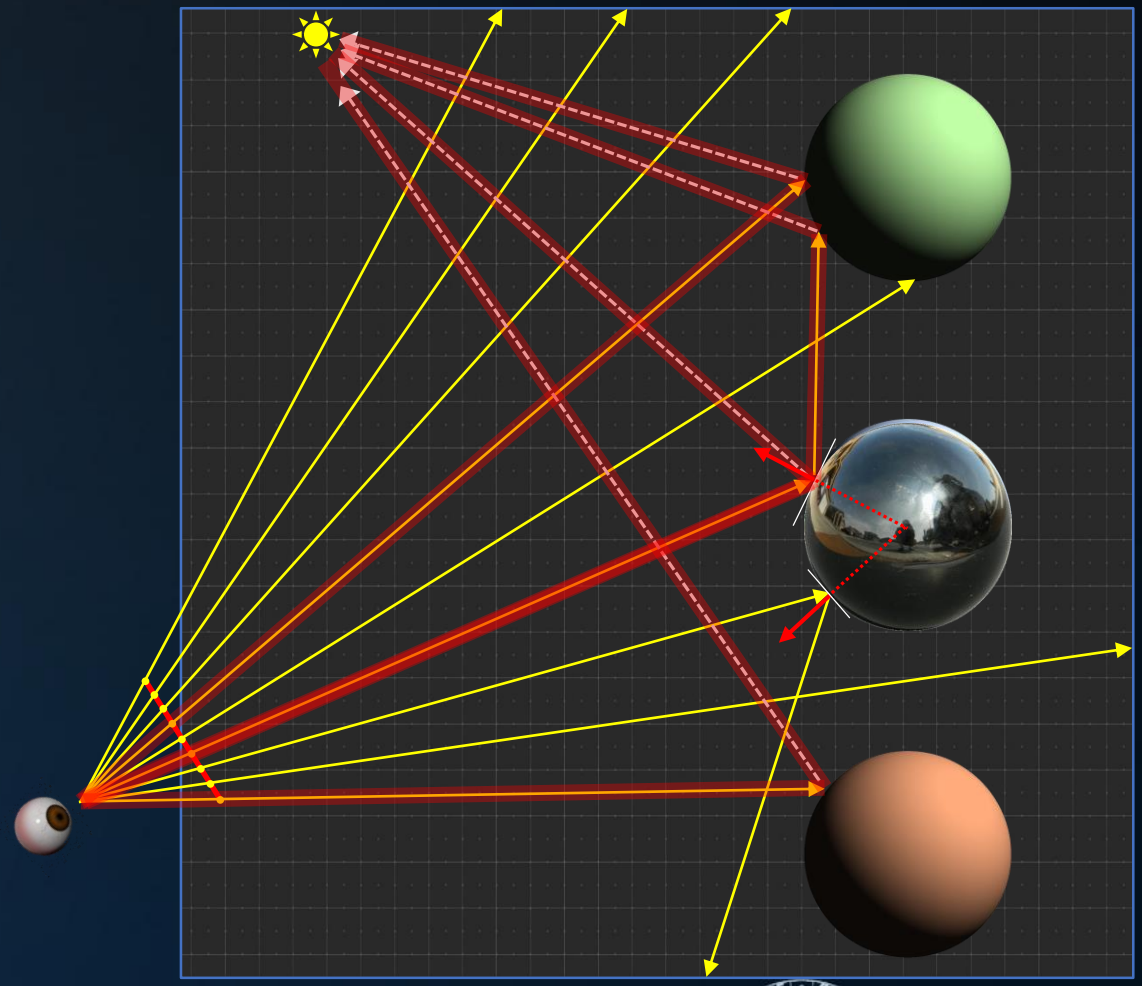
Ray setup

```
void Render( const mat4& T )
{
   vec3 p0 = T * vec3( -1,  1,  1 );   // top-left
   vec3 p1 = T * vec3(  1,  1,  1 );   // top-right
   vec3 p2 = T * vec3( -1, -1,  1 );   // bottom-left
   vec3 E = T * vec3( 0, 0, 0 );
   vec3 right = p1 – p0;
   vec3 down = p2 – p0;
   for( int y = 0; y < 512; y++ )
   {
      for( int x = 0; x < 512; x++ )
      {
         float u = x / 512.f;
         float v = y / 512.f;
         vec3 P = p0 + u * right + v * down;
         vec3 D = normalize( P – E );
         vec3 c = Trace( Ray( E, D ) );
         // TODO: visualize c somehow
      }
   }
}
```



Ray setup

```
void Render( const mat4& T )
{
    vec3 p0 = T * vec3( -1,  1,  1 );
    vec3 p1 = T * vec3(  1,  1,  1 );
    vec3 p2 = T * vec3( -1, -1,  1 );
    vec3 E = T * vec3( 0, 0, 0 );
    vec3 right = p1 - p0;
    vec3 down = p2 - p0;
    for( int y = 0; y < 512; y++ )
    {
        for( int x = 0; x < 512; x++ )
        {
            float u = x / 512.f;
            float v = y / 512.f;
            vec3 P = p0 + u * right + v * down;
            vec3 D = normalize( P - E );
            vec3 c = Trace( Ray( E, D ) );
            // TODO: visualize c somehow
        }
    }
}
```



Trace

Needful things:

Ray/plane intersection:

Plane: $P \cdot \vec{N} + d = 0$

Ray: $P(t) = O + t\vec{D}$

Substituting for $P(t)$, we get: $t = -(O \cdot \vec{N} + d)/(\vec{D} \cdot \vec{N})$

Ray/sphere intersection:

```
void Sphere::IntersectSphere( Ray ray )
{
    vec3 C = this.pos - ray.O;
    float t = dot( C, ray.D );
    vec3 Q = C - t * ray.D;
    float p2 = dot( Q, Q );
    if (p2 > sphere.r2) return; // r2 = r * r
    t -= sqrt( sphere.r2 - p2 );
    if ((t < ray.t) && (t > 0)) ray.t = t;
}
```

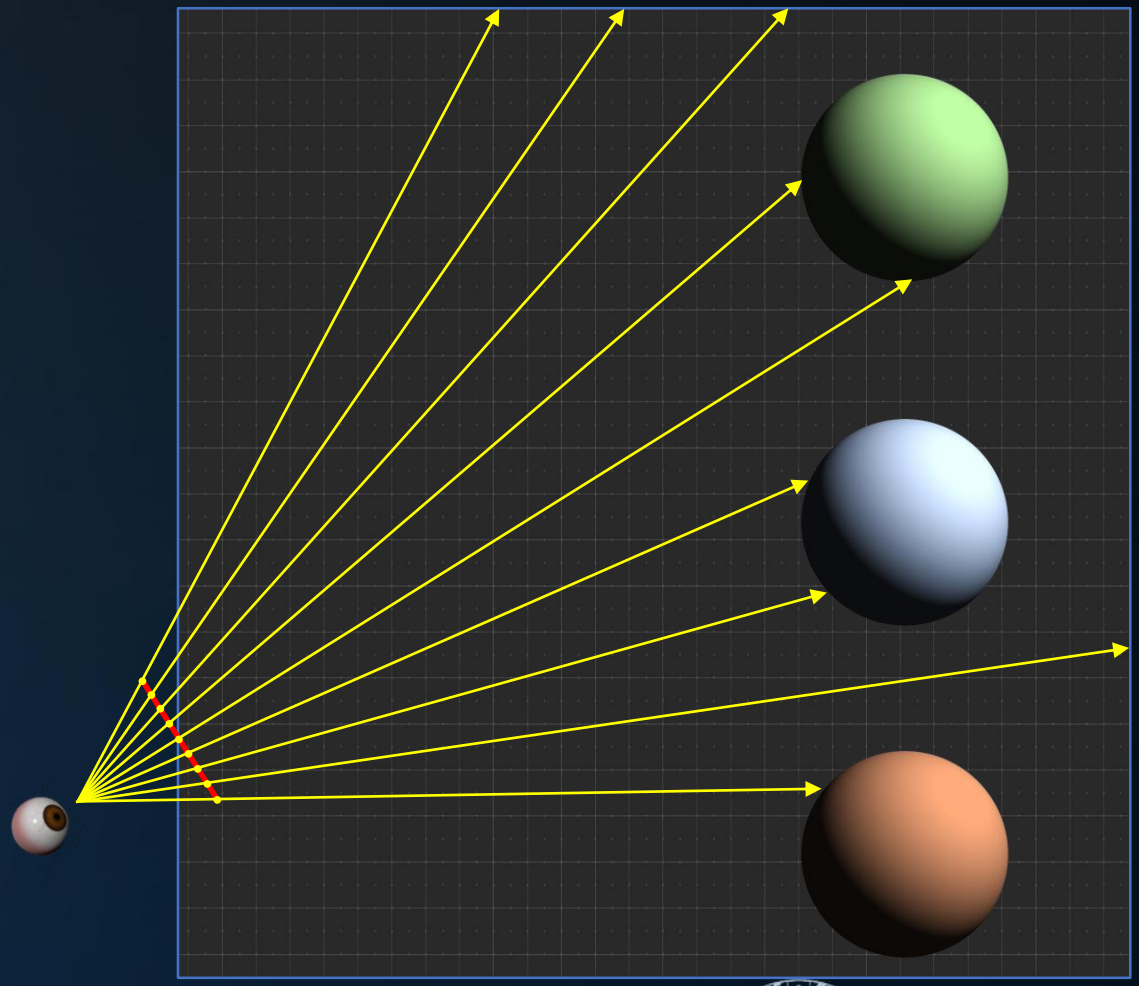
Ray/triangle intersection:

Adapt from Moller-Trumbore, via Wikipedia:

```
bool RayIntersectsTriangle(vec3 rayOrigin, vec3 rayVector, Triangle* tri,vec3& intersection)
{
    const float EPSILON = 0.0000001;
    vec3 vertex0 = inTriangle->vertex0;
    vec3 vertex1 = inTriangle->vertex1;
    vec3 vertex2 = inTriangle->vertex2;
    vec3 edge1, edge2, h, s, q;
    float a,f,u,v;
    edge1 = vertex1 - vertex0, edge2 = vertex2 - vertex0;
    h = rayVector.crossProduct(edge2);
    a = edge1.dotProduct(h);
    if (a > -EPSILON && a < EPSILON) return false; // ray is parallel to this triangle
    f = 1.0/a, s = rayOrigin - vertex0, u = f * s.dotProduct(h);
    if (u < 0.0 || u > 1.0) return false; // miss
    q = s.crossProduct(edge1), v = f * rayVector.dotProduct(q);
    if (v < 0.0 || u + v > 1.0) return false; // miss
    float t = f * edge2.dotProduct(q);
    if (t > EPSILON && t < 1/EPSILON) // ray intersection
    {
        intersection = rayOrigin + rayVector * t;
        return true;
    }
    else return false; // miss
}
```

Ray setup

```
void Render( const mat4& T )
{
    vec3 p0 = T * vec3( -1,  1,  1 );
    vec3 p1 = T * vec3(  1,  1,  1 );
    vec3 p2 = T * vec3( -1, -1,  1 );
    vec3 E = T * vec3( 0, 0, 0 );
    vec3 right = p1 – p0;
    vec3 down = p2 – p0;
    for( int y = 0; y < 512; y++ )
    {
        for( int x = 0; x < 512; x++ )
        {
            float u = x / 512.f;
            float v = y / 512.f;
            vec3 P = p0 + u * right + v * down;
            vec3 D = normalize( P – E );
            vec3 c = Trace( Ray( E, D ) );
            // TODO: visualize c somehow
        }
    }
}
```



Whitted-style Ray Tracing

```
vec3 Trace( const Ray& r )
{
   I, N, material = NearestIntersection( scene, r );
   return material.color * DirectIllumination( I, N );
}
```

Direct illumination:

Summed contribution of *unoccluded* point light sources, taking into account:

- Distance to I
- Angle between $\vec{L}$ and $\vec{N}$
- Intensity of light source

Note that this requires a ray per light source.

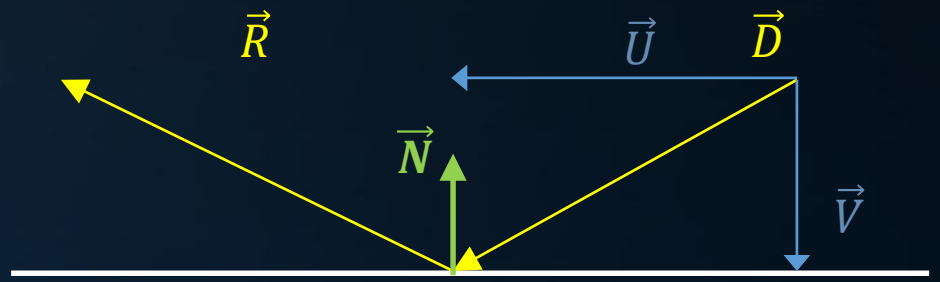
Whitted-style Ray Tracing

```
vec3 Trace( const Ray& r )
{
   I, N, material = NearestIntersection( scene, r );
   if (material.IsDiffuse())
      return material.color * DirectIllumination( I, N );
   else
      return material.color * Trace( … );
}
```

Reflection:

$$\vec{R} = \vec{D} - 2(\vec{D} \cdot \vec{N})\vec{N}$$

$$\vec{V} = \vec{N}(\vec{D} \cdot \vec{N})$$
$$\vec{U} = \vec{D} - \vec{V}$$
$$\vec{R} = \vec{U} + (-\vec{V})$$
$$\vec{R} = \vec{D} - \vec{N}(\vec{D} \cdot \vec{N}) - \vec{N}(\vec{D} \cdot \vec{N})$$
$$\vec{R} = \vec{D} - 2(\vec{D} \cdot \vec{N})\vec{N}$$



Dielectrics

The direction of the transmitted vector $\vec{T}$ depends on the refraction indices n_1, n_2 of the media separated by the surface. According to Snell's Law:

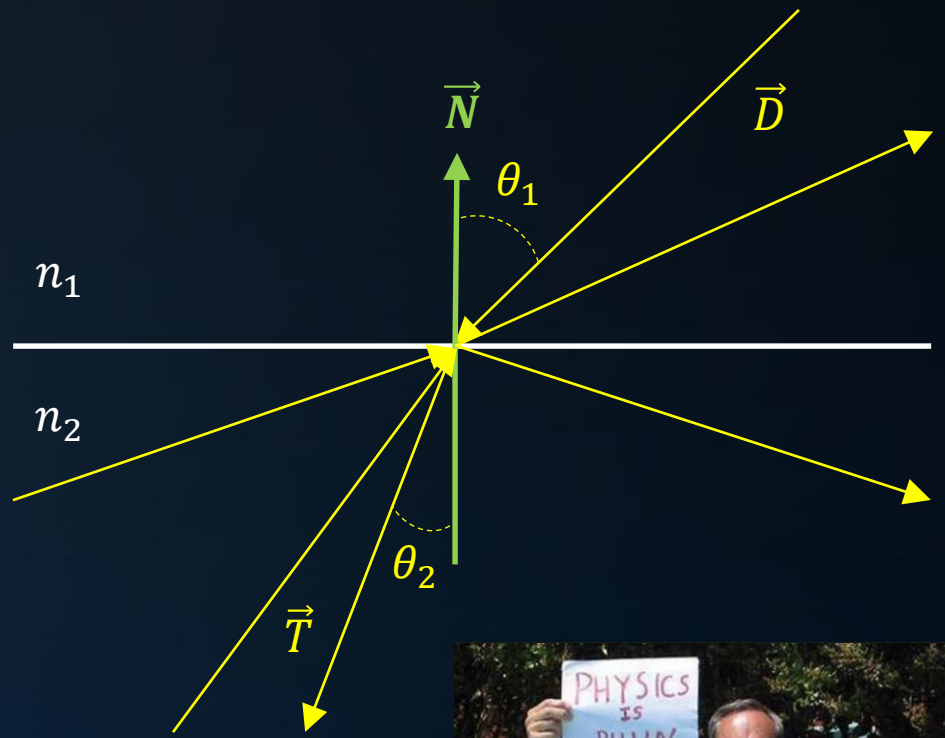
$$n_1 sin\theta_1 = n_2 sin\theta_2$$

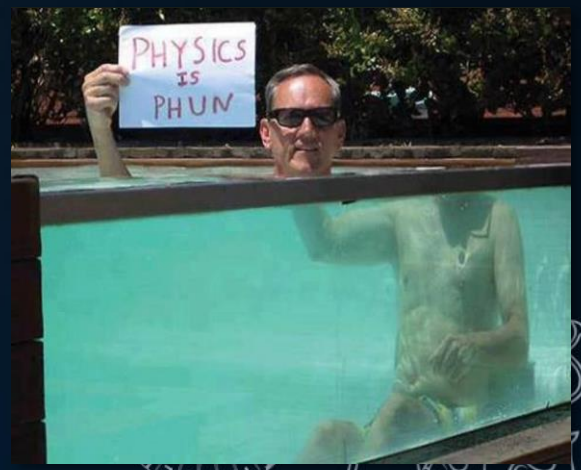
or

$$\frac{n_1}{n_2} sin\theta_1 = sin\theta_2$$

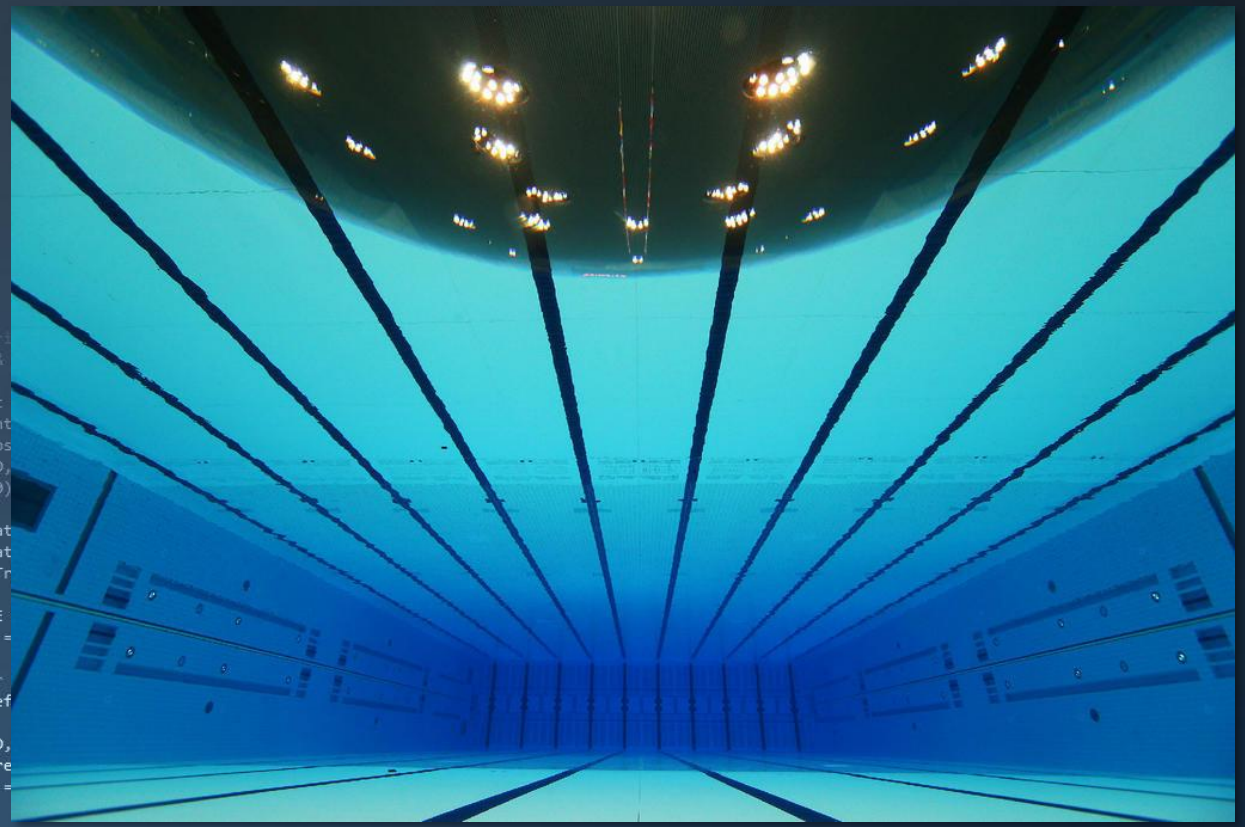
Note: left term may exceed 1, in which case θ_2 cannot be computed. Therefore:

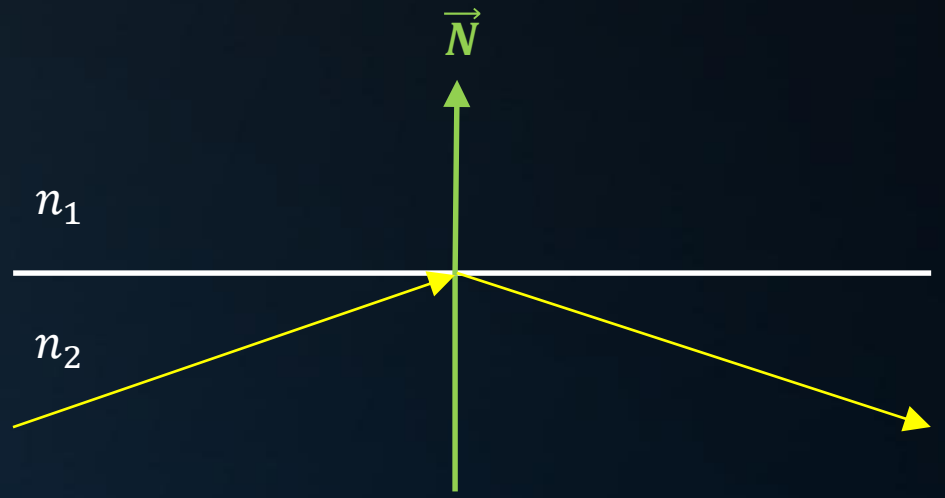
$$\frac{n_1}{n_2} sin\theta_1 = sin\theta_2 \Leftrightarrow sin\theta_1 \leq \frac{n2}{n1} \rightarrow \theta_{critical} = \arcsin\left(\frac{n_2}{n_1}\sin\theta_2\right)$$





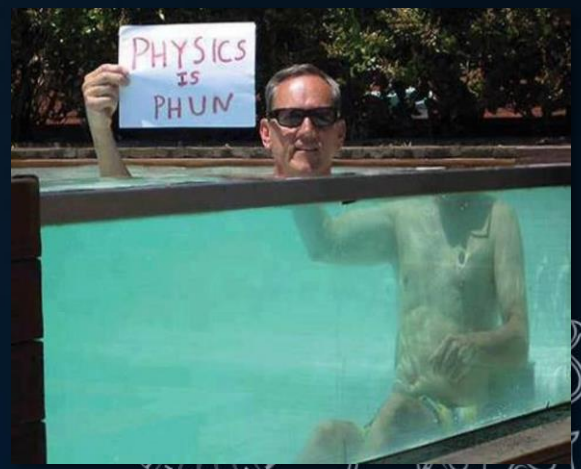
Ray Tracing for Games





Note: left term may exceed 1, in which case θ_2 cannot be computed. Therefore:

$$\frac{n_1}{n_2} sin\theta_1 = sin\theta_2 \Leftrightarrow sin\theta_1 \leq \frac{n2}{n1} \rightarrow \theta_{critical} = \arcsin\left(\frac{n_2}{n_1}\sin\theta_2\right)$$

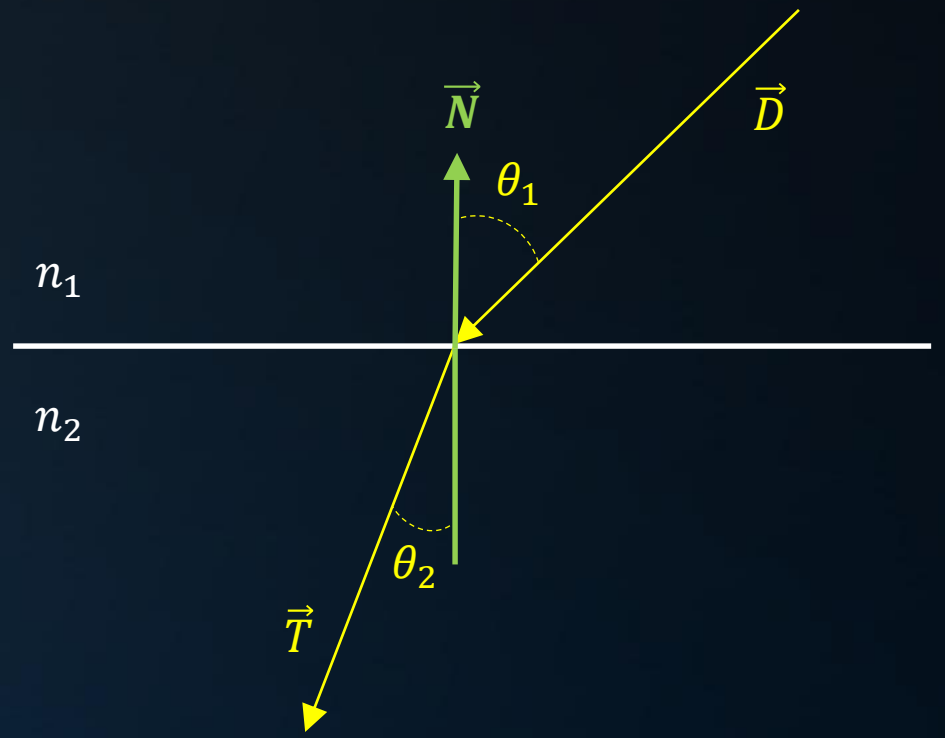


Dielectrics

$$\frac{n_1}{n_2} sin\theta_1 = sin\theta_2 \Leftrightarrow sin\theta_1 \leq \frac{n2}{n1}$$

$$k = 1 - \left(\frac{n_1}{n_2}\right)^2 (1 - cos{\theta_1}^2)$$

$$\vec{T} = \begin{cases} TIR, & for\ k < 0 \\ \frac{n_1}{n_2}\vec{D} + \vec{N}\left(\frac{n_1}{n_2} cos\theta_1 - \sqrt{k}\right), & for\ k \geq 0 \end{cases}$$

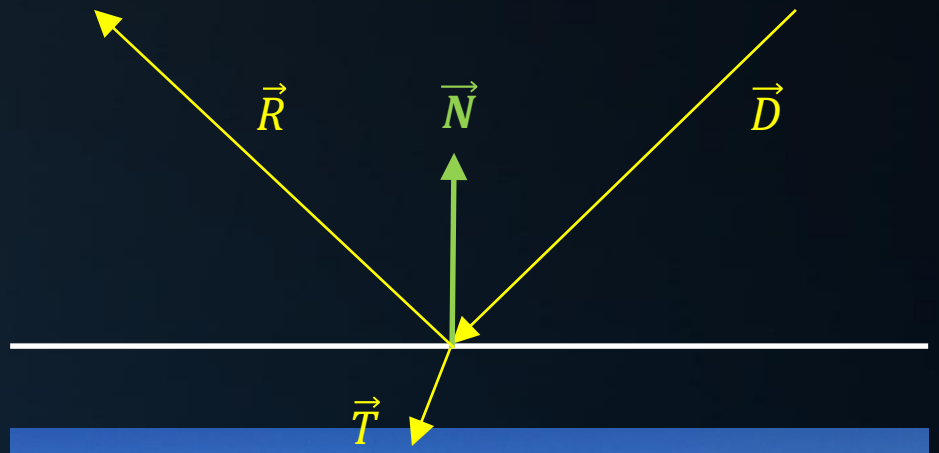


Note: $cos\theta_1 = \vec{N} \cdot -\vec{D}$, and $\frac{n_1}{n_2}$ should be calculated only once.

* For a full derivation, see http://www.flipcode.com/archives/reflection_transmission.pdf

Dielectrics

A typical dielectric transmits *and* reflects light.



Dielectrics

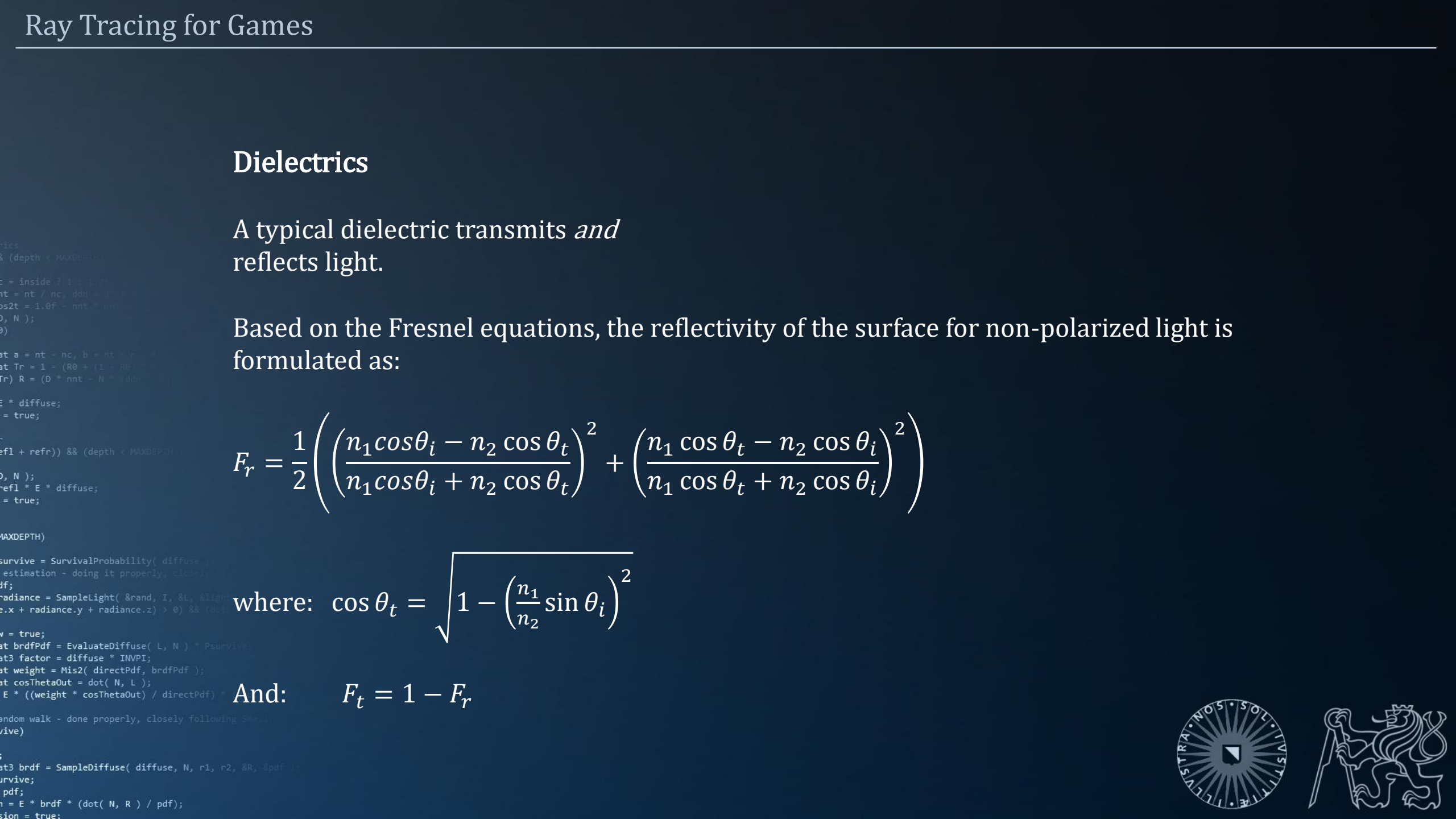
A typical dielectric transmits *and* reflects light.

Based on the Fresnel equations, the reflectivity of the surface for non-polarized light is formulated as:

$$F_r = \frac{1}{2}\left(\left(\frac{n_1 cos\theta_i - n_2 \cos\theta_t}{n_1 cos\theta_i + n_2 \cos\theta_t}\right)^2 + \left(\frac{n_1 \cos\theta_t - n_2 \cos\theta_i}{n_1 \cos\theta_t + n_2 \cos\theta_i}\right)^2\right)$$

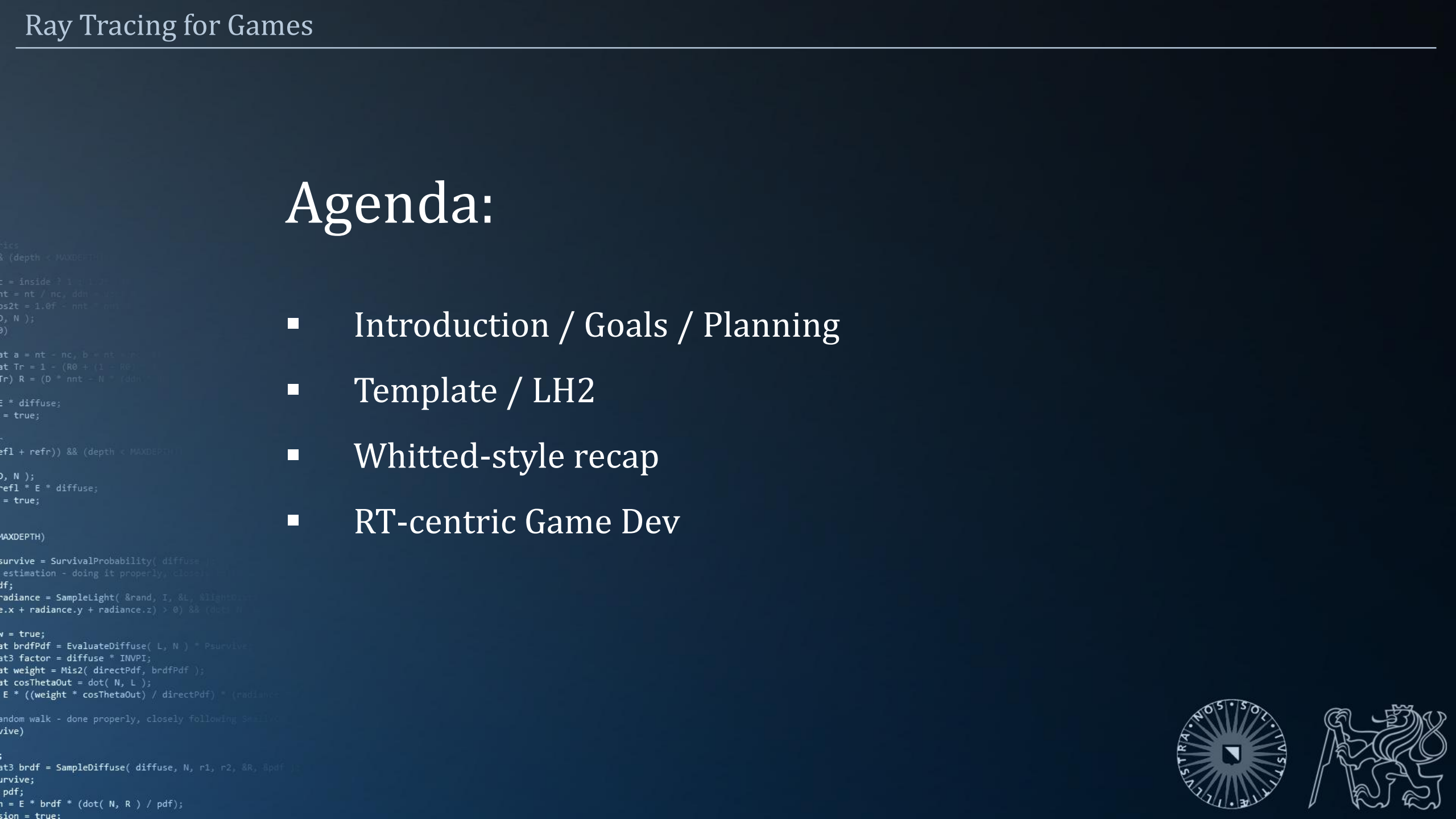
where: $\cos\theta_t = \sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2}$

And: $F_t = 1 - F_r$



Agenda:

- Introduction / Goals / Planning
- Template / LH2
- Whitted-style recap
- RT-centric Game Dev



Ray Tracing for Games

Games are often built to showcase technology.

Wolfenstein: indoor, no detail on the floors, sprites for enemies.

Ray Tracing for Games

Games are often built to showcase technology.

Doom: indoor, height levels, horizontal floors.

Ray Tracing for Games

Games are often built to showcase technology.

Quake: textures everywhere, polygon enemies, skybox. Still indoor.

Games are often built to showcase technology.

Quake 2: colored lights, more detail, some outdoor parts.

Ray Tracing for Games

Games are often built to showcase technology.

Unreal 1: much more outdoor gameplay. Particles. Shiny floors.

Ray Tracing for Games

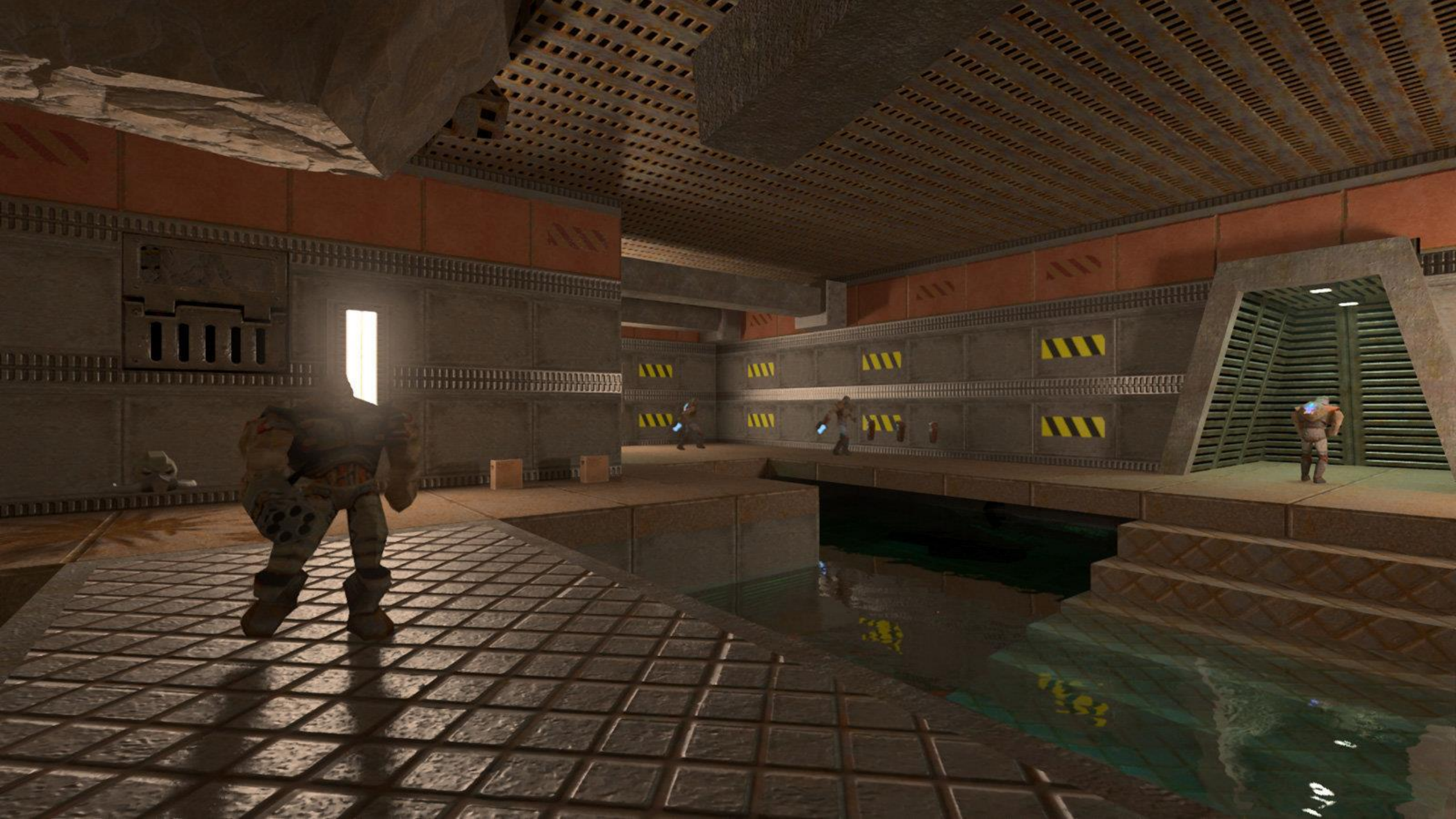
Games are often built to showcase technology.

Duke Nuke'm: camera can't tilt up and down (well, not really).

Ray Tracing for Games

Games are often built to showcase technology.

FarCry: instancing (trees). Almost everything outside.




AKER
MAGAZIJN
P


La Casa Mexicana
NVIDIA
GEFORCE RTX

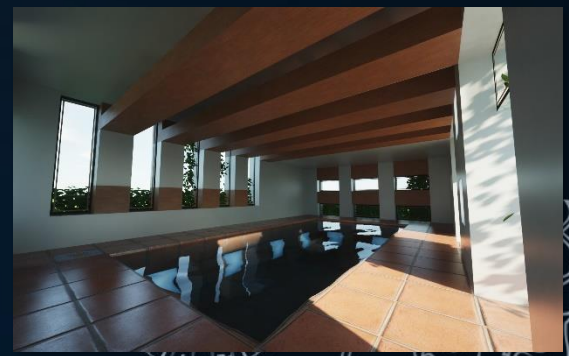
A Ray Traced Game

Strengths of ray tracing:

- Correct shadows
- Correct reflection
- Correct refraction
- …

Not so strong:

- Lots of animation
- High resolution
- High frame rate



1997
3Dfx

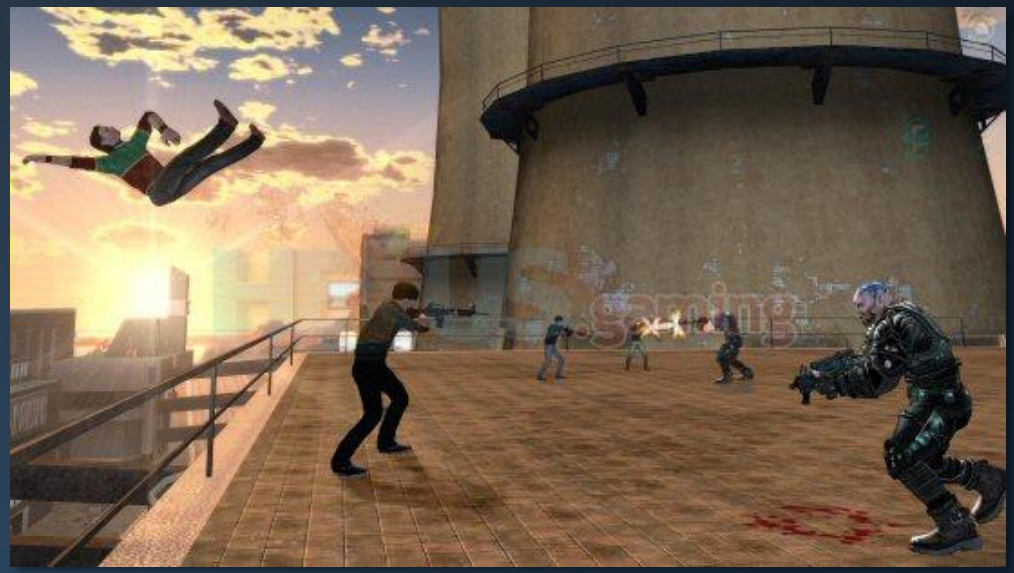

4
86
0:04.67
8/8
1/4

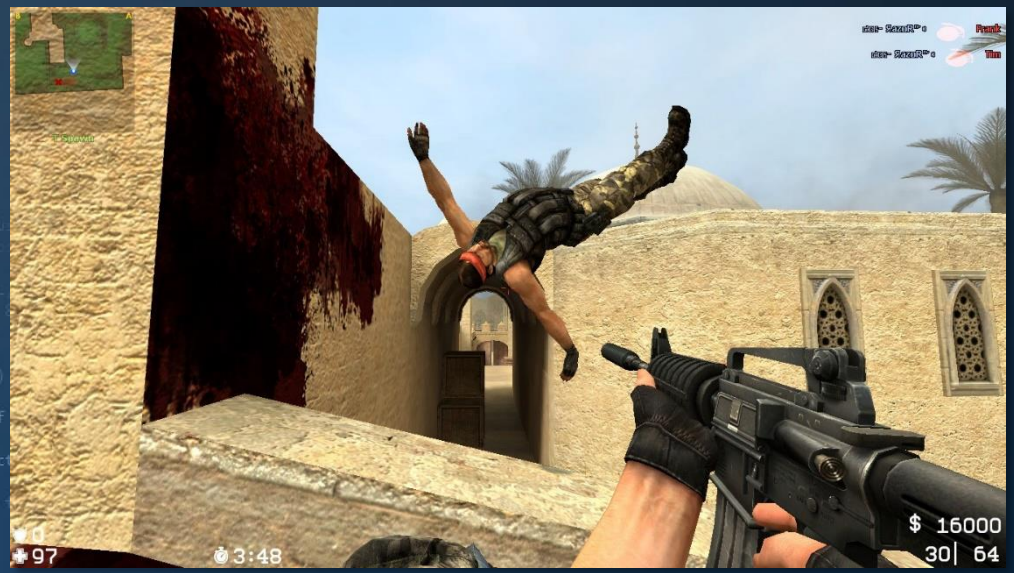

DOSGamers.com

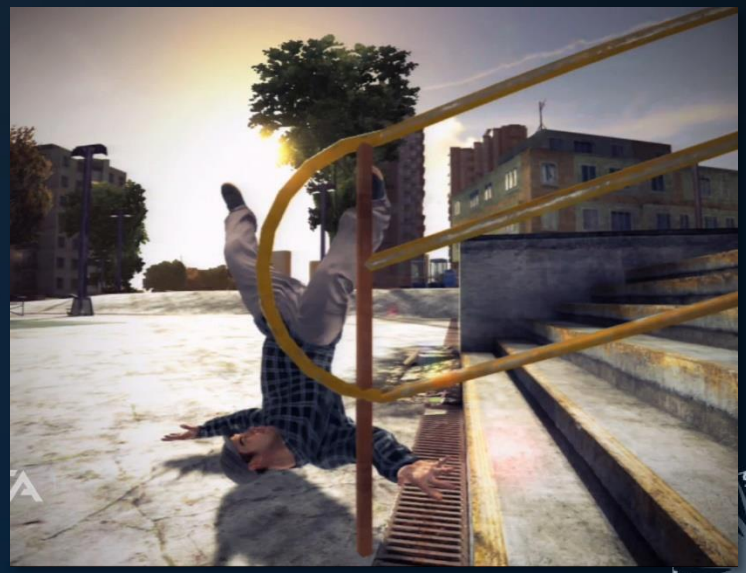

2/3
LAP
4:27.94
TIME
1/12
POS
665


CP 1/8 LAP 1/2
WASTED 0/5
2:56
0 PROFIT
1/527 KILLS
A
P
0
2 053

Ray Tracing for Games

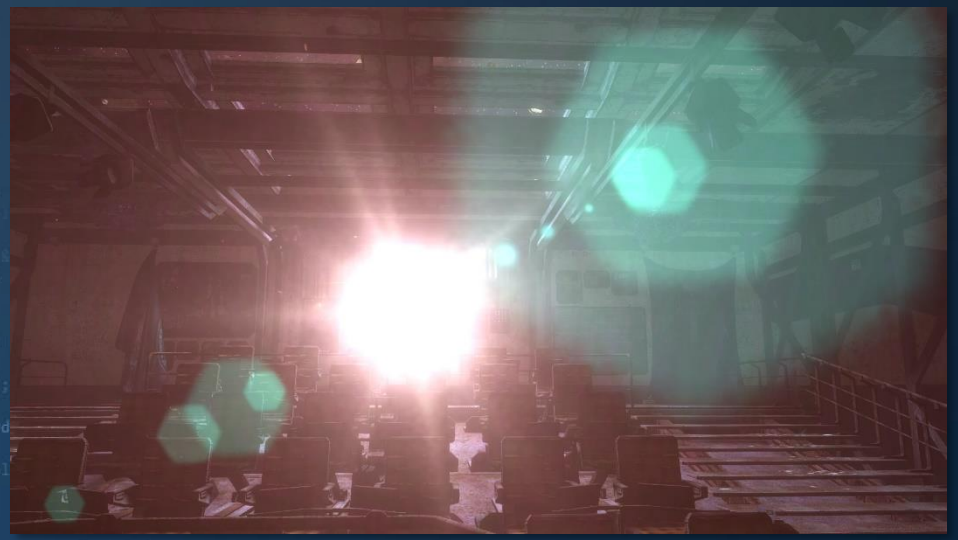






1999
Ragdoll
physics

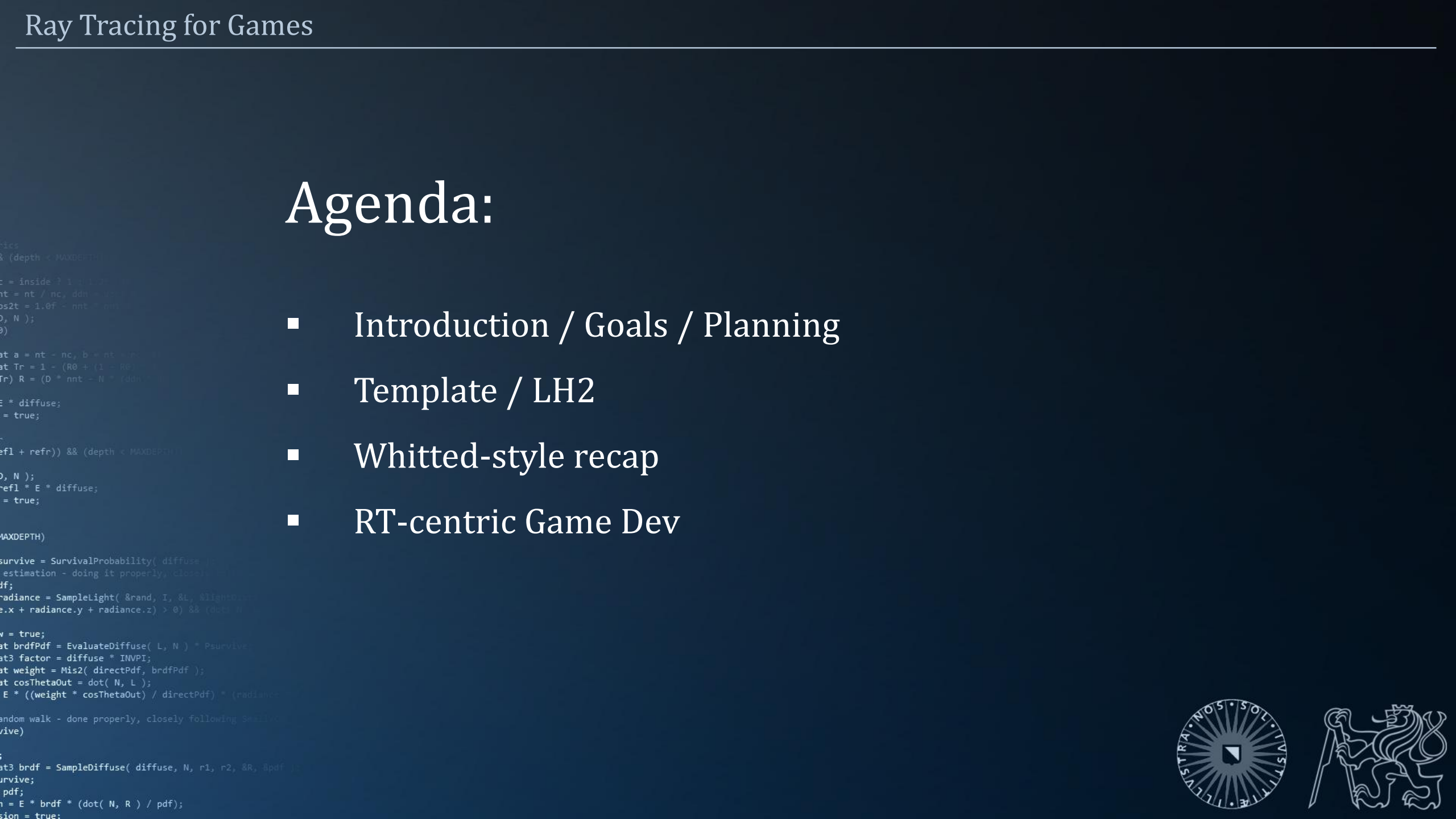
2005 lens flares



Ray Tracing for Games

Agenda:

- Introduction / Goals / Planning
- Template / LH2
- Whitted-style recap
- RT-centric Game Dev



Ray Tracing for Games

Wednesday
13:00 – 17:00

course intro
LH2
template
Whitted
refactoring
RT-centric games

LAB 1

Thursday
09:00 – 14:00

advanced Whitted
audio, AI & physics
faster Whitted
Heaven7

LAB 2

work @ home

End result day 2:

A solid Whitted-style ray tracer, as a basis for subsequent work.

Friday
09:00 – 17:00

optimization
profiling, rules of engagement
threading

LAB 3

SIMD
applied SIMD
SIMD triangle
SIMD AABB

LAB 4

End result day 3:

A 5x faster tracer.

GLOBAL GAME JAM

End product:

1. Path tracer that produces pretty images in a few minutes.
2. Real-time ray tracer on the CPU or GPU.
3. Raytraced game or demo.
4. RTX port of an existing game.

End of PART 1.