Today

• **Rendering**
  - Real-time (What’s the best you can do with low-computation budget?)
  - Materials and Lighting
  - Offline (How fast can you compute as-realistic-as-desired images?)

• **Geometry & Simulation**
  - Understand meshes & discrete representations
  - Understand « physics »
Ray tracing basic principle

\[ L(p, \omega_{\text{out}}) = L_e(p, \omega_{\text{out}}) + \int_{\omega_{\text{in}} \in \mathcal{H}(p, n)} L(p, \omega_{\text{in}}) \ast \text{brdf}(p, \omega_{\text{in}}, \omega_{\text{out}}) \cdot (n \cdot \omega_{\text{in}}) \ . d\omega_{\text{in}} \]
Ray tracing basic principle

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p, n)} L(p, \omega_{in}) \ast brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Ray tracing basic principle

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p, n)} L(p, \omega_{in}) \times brdf(p, \omega_{in}, \omega_{out}) \times (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Ray tracing basic principle

\[ L(p, \omega_{\text{out}}) = L_e(p, \omega_{\text{out}}) + \int_{\omega_{\text{in}} \in \mathcal{H}(p,n)} L(p, \omega_{\text{in}}) \ast \text{brdf}(p, \omega_{\text{in}}, \omega_{\text{out}}) \cdot (n \cdot \omega_{\text{in}}) \ . d\omega_{\text{in}} \]
Ray tracing basic principle

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast brdf(p, \omega_{in}, \omega_{out}). \left( n \cdot \omega_{in} \right) \cdot d\omega_{in} \]
Ray tracing basic principle

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) * brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Ray tracing basic principle

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) * \text{brdf}(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Ray tracing basic principle

\[ L(p, \omega_{\text{out}}) = L_e(p, \omega_{\text{out}}) + \int_{\omega_{\text{in}} \in \mathcal{H}(p,n)} L(p, \omega_{\text{in}}) \ast \text{brdf}(p, \omega_{\text{in}}, \omega_{\text{out}}) \cdot \langle n, \omega_{\text{in}} \rangle \, d\omega_{\text{in}} \]
Ray tracing basic principle

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) * brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Ray tracing basic principle

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) \quad + \quad \int_{\omega_{in} \in H(p, n)} L(p, \omega_{in}) * brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Ray tracing basic principle

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) * brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Ray tracing basic principle

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Ray tracing basic principle

• **Rebounds**
  - How many rays at each intersection?
  - When do we stop?

• **Implementation**
  - Recursive vs path-based
  - Ray/Scene intersection

• **Simple effects « for free »**
  - Mirrors/Reflections
  - Glass/Refractions, Caustics
  - ...
Reflections
Reflections

What do you need to change?

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p, n)} L(p, \omega_{in}) \ast brdf(p, \omega_{in}, \omega_{out}) \cdot \left( \overrightarrow{n \cdot \omega_{in}} \right) \cdot d\omega_{in} \]
Reflections

\[
L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} \left( p, \omega_{in} \right) \cdot \left( \omega_{in}, \omega_{out} \right) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in}
\]
Refractions

Law of sines
Refractions
Refractions

What do you need to change?

\[
L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in}
\]
Refractions

Add rays on the other side (percentage is controlled by the transparency)

\[
L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast \text{brdf}(p, \omega_{in}, \omega_{out}) \cdot \frac{(n \cdot \omega_{in})}{\text{produit scalaire}} \ . d\omega_{in}
\]
Refractions

Create caustics
Depth of field
What to change?

Eye (camera center)

Image plane

Pixel

Ray
What to change?

aperture

f

Focal plane

ray
New ray from random point in aperture

Careful! You still add the contribution of the ray to the original pixel!
Spatial hierarchies: kd-trees

- Binary tree of space subdivisions
  - Each is axis-aligned plane
Spatial hierarchies: kd-trees

- Binary tree of space subdivisions
  - Each is axis-aligned plane
Spatial hierarchies: kd-trees

- Binary tree of space subdivisions
  - Each is axis-aligned plane
Spatial hierarchies: kd-trees

- Binary tree of space subdivisions
  - Each is axis-aligned plane
Spatial hierarchies: kd-trees

- Binary tree of space subdivisions
  - Each is axis-aligned plane

Question:
How to decide where to cut?
Question:
In which node do we put a triangle that intersects the cutting plane?
Ray - kdtree intersection

- Traversing a kd-tree: recursive
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
      - If ray intersects both children, recurse on near side, then far side
      - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
Ray - kd-tree intersection

- Traversing a kd-tree: recursive
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
      - If ray intersects both children, recurse on near side, then far side
      - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
Ray - kdtree intersection

- Traversing a kd-tree: recursive
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
      - If ray intersects both children, recurse on near side, then far side
      - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
Ray - kd-tree intersection

- Traversing a kd-tree: recursive
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
      - If ray intersects both children, recurse on near side, then far side
      - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
Ray – kdtree intersection

- Traversing a kd-tree: recursive
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
      - If ray intersects both children, recurse on near side, then far side
      - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
Ray - kdtree intersection

- **Traversing a kd-tree: recursive**
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
      - If ray intersects both children, recurse on near side, then far side
      - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
Traversing a kd-tree: recursive
- Start at root node
- For current node:
  - If inner node:
    - Find intersection of ray with plane
    - If ray intersects both children, recurse on near side, then far side
    - Otherwise, recurse on side it intersects
  - If leaf node:
    - Intersect with all object. If hit, terminate.
Ray – kdtree intersection

- Traversing a kd-tree: recursive
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
      - If ray intersects both children, recurse on near side, then far side
      - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
Ray - kdtree intersection

- Traversing a kd-tree: recursive
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
      - If ray intersects both children, recurse on near side, then far side
      - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
Ray - kd-tree intersection

- Traversing a kd-tree: recursive
  - Start at root node
  - For current node:
    - If inner node:
      - Find intersection of ray with plane
        - If ray intersects both children, recurse on near side, then far side
        - Otherwise, recurse on side it intersects
    - If leaf node:
      - Intersect with all object. If hit, terminate.
Kd-tree traversal

- Simple and fast implementation
  - In practice: using stack, not recursion
  - Very quick intersection test (couple FLOPS + tests)

- Overall: logarithmic complexity for each ray
Recursive (pseudo) implementation

```
Vec3 Scene ::rayTrace( Ray r ) {
    if( intersectLight(r) )
        return lightColor ;
    Vec3 color(0,0,0) ;
p,n = intersectScene( r ) ;
for( int i = 0 ; i < N ; ++i ){
    Ray r2 = randomRay( p , n ) ;
    color += brdf( p , n , -r , r2 ) * rayTrace( r2 ) * dr2 ;
}
return color ;
}
```

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \cdot \text{brdf}(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Recursive (pseudo) implementation

```cpp
Vec3 Scene::rayTrace(Ray r) {
    if (intersectLight(r))
        return lightColor;
    Vec3 color(0,0,0);
    p,n = intersectScene(r);
    for (int i = 0; i < N; ++i) {
        Ray r2 = randomRay(p,n);
        color += brdf(p,n,-r,r2) * rayTrace(r2) * dr2;
    }
    return color;
}
```

\[
L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) * brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) . d\omega_{in}
\]
Recursive (pseudo) implementation

Vec3 Scene ::rayTrace( Ray r ) {
    if( intersectLight(r) )
        return lightColor ;
    Vec3 color(0,0,0) ;
    p,n = intersectScene( r ) ;
    for( int i = 0 ; i < N ; ++i ){
        Ray r2 = randomRay( p , n ) ;
        color += brdf( p , n , -r , r2 ) * rayTrace( r2 ) * dr2 ;
    }
    return color ;
}

\[
L(p,\omega_{out}) = L_e(p,\omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p,\omega_{in}) \ast brdf(p,\omega_{in},\omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in}
\]
Recursive (pseudo) implementation

```
Vec3 Scene::rayTrace(Ray r) {
  if( intersectLight(r) )
    return lightColor;
  Vec3 color(0,0,0);
p, n = intersectScene(r);
  for( int i = 0; i < N; ++i ){
    Ray r2 = randomRay(p, n);
    color += brdf(p, n, -r, r2) * rayTrace(r2) * dr2;
  }
  return color;
}
```

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast \text{brdf}(p, \omega_{in}, \omega_{out}). \left( n \cdot \omega_{in} \right) \cdot d\omega_{in} \]
Recursive (pseudo) implementation

```c
Vec3 Scene ::rayTrace( Ray r ) {
    if( intersectLight(r) )
        return lightColor ;
    Vec3 color(0,0,0) ;
    p,n = intersectScene( r ) ;
    for( int i = 0 ; i < N ; ++i ){
        Ray r2 = randomRay( p , n ) ;
        color += brdf( p , n , -r , r2 ) * rayTrace( r2 ) * dr2 ;
    }
    return color ;
}
```

$$L(p,\omega_{out}) = L_e(p,\omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p,\omega_{in}) * brdf(p,\omega_{in},\omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in}$$
Non-recursive (pseudo) implementation

Vec3 Scene::rayTrace( Ray r ) {
    Vec3 color(0,0,0);
    for( int i = 0 ; i < N ; ++i ){
        Path p = randomPath( r ) ;
        Vec3 pathColor(1,1,1);
        for( int j = 0 ; j < p.depth() ; ++j )
            pathColor *= brdf( p.in(j) , p.out(j) ) * p.material(j) ;
        color += pathColor * p.light() * dp
    }
    return color ;
}

\[
L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast \text{brdf}(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \ast \text{produit scalaire} . d\omega_{in}
\]
Non-recursive (pseudo) implementation

```cpp
Vec3 Scene :: rayTrace( Ray r ) {
    Vec3 color(0,0,0);
    for( int i = 0 ; i < N ; ++i ){
        Path p = randomPath( r );
        Vec3 pathColor(1,1,1);
        for( int j = 0 ; j < p.depth() ; ++j )
            pathColor *= brdf( p.in(j), p.out(j) ) * p.material(j);
        color += pathColor * p.light() * dp
    }
    return color;
}
```

\[
L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \cdot \text{brdf}(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in}
\]
Non-recursive (pseudo) implementation

```cpp
Vec3 Scene :: rayTrace(Ray r) {
    Vec3 color(0,0,0);
    for( int i = 0 ; i < N ; ++i ){
        Path p = randomPath( r );
        Vec3 pathColor(1,1,1);
        for( int j = 0 ; j < p.depth() ; ++j )
            pathColor *= brdf( p.in(j), p.out(j) ) * p.material(j);
        color += pathColor * p.light() * dp;
    }
    return color;
}
```

$$L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \cdot brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in}$$
Non-recursive (pseudo) implementation

Vec3 Scene::rayTrace(Ray r) {
    Vec3 color(0,0,0);
    for( int i = 0 ; i < N ; ++i ){
        Path p = randomPath(r);
        Vec3 pathColor(1,1,1);
        for( int j = 0 ; j < p.depth() ; ++j )
            pathColor *= brdf(p.in(j), p.out(j)) * p.material(j);
        color += pathColor * p.light() * dp
    }
    return color;
}
Recursive vs path-based

• **Recursive**:
  - Easy to implement (see [here](#) for example)
  - Requires casting « shadow rays » in practice

• **Path-based**
  - More complicated
  - Allows for parallelization
  - Allows for useful extensions:
    • Bidirectional
    • Metropolis
Importance sampling

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Importance sampling

\[ L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p, n)} L(p, \omega_{in}) \ast \text{brdf}(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in} \]
Importance sampling

More rays here:
Requires to set a smaller step size $dw_i$

$$L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast brdf(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in}$$
Importance sampling

\[
\int f(x) \, dx = \int \frac{f(x)}{p(x)} \, p(x) \, dx
\]

\[
\int f(x) \, dx = \frac{1}{N} \sum_i \frac{f(x_i)}{p(x_i)}
\]

if \( x_i \) follows the distribution \( p \)

\[
L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast \text{brdf}(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in}
\]

produit scalaire
Importance sampling

In practice, almost everything is importance-sampled (lights, bounces, ...). Variance is best reduced when p equals f.

\[
L(p, \omega_{out}) = L_e(p, \omega_{out}) + \int_{\omega_{in} \in \mathcal{H}(p,n)} L(p, \omega_{in}) \ast \text{brdf}(p, \omega_{in}, \omega_{out}) \cdot (n \cdot \omega_{in}) \cdot d\omega_{in}
\]
Bidirectional path tracing principle

- It is sometimes difficult to hit the lights (small lights, occluders) → slow convergence
Bidirectional path tracing principle

- It is sometimes difficult to hit the lights (small lights, occluders) → slow convergence
- We can generate sub-paths from the lights
Bidirectional path tracing principle

- It is sometimes difficult to hit the lights (small lights, occluders) → slow convergence
- We can generate sub-paths from the lights, and sub-paths from the camera
Bidirectional path tracing principle

- It is sometimes difficult to hit the lights (small lights, occluders) → slow convergence
- We can generate sub-paths from the lights, and sub-paths from the camera
- And connect them
Bidirectional path tracing principle

(a) unidirectional

(b) bidirectional
Bidirectional path tracing principle

(a) unidirectional
(b) bidirectional