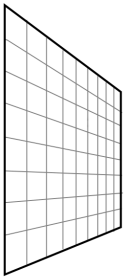
Ray Tracing & Global Illumination

Ray Tracing

Kai Selgrad

Winterterm 2021



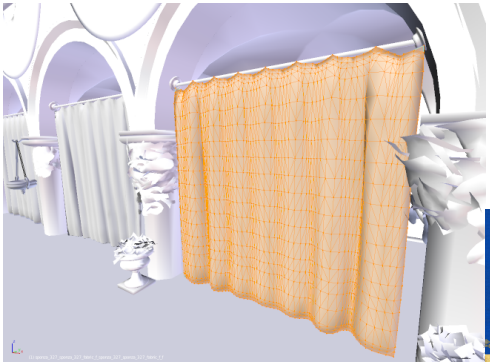
Basic CG terminology

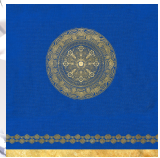
- Scenes consist of primitives
- Triangles are a very common primitive
- Images consist of pixels
- Pixels are made up of color triplets
- Each color component (channel) ist usually in the range $[0, 1]$
- This range is most often implemented by a single byte, mapped accordingly
- High Dynamic Range Rendering: $[0, \infty]$ with floating point representation (needs projection to screen's range at the end)
- The 'camera' specifies the viewer configuration

Common representation for positions, directions and colors in RTGI code:

```
struct vec3 {
    float x, y, z;
};
```

Meshes, Triangles & Vertices





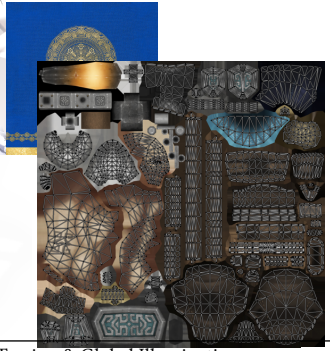
More terminology

- Meshes are an approximation of (real world or imagined) objects
- In RTGI all meshes are merged to a big set of primitives
- Each primitive references its material and vertices
- Each vertex holds position, normal and texture coordinates

Meshes, Triangles & Vertices

More terminology

- Meshes are an approximation of (real world or imagined) objects
- In RTGI all meshes are merged to a big set of primitives
- Each primitive references its material and vertices
- Each vertex holds position, normal and texture coordinates



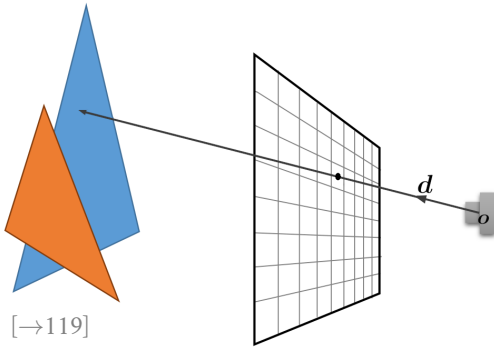
Ray Casting

The basic ray casting outline is

- Set up ray
  - Camera position is at $\boldsymbol{o}$
  - Connect the camera position with a point on a pixel
  - Normalize to get the direction to that point $\boldsymbol{d}$
  - Ray $r = \boldsymbol{o} + t\boldsymbol{d}$

To obtain $\boldsymbol{d}$ we have to find a pixel position in our 3D scene, i.e. we have to set up a mapping between our world and our target image.

To this end we imagine the image as a plane in front of the camera, one unit along its main viewing direction $\boldsymbol{e}$, such that the plane covers the area spanned by the field of view angles.



[→119]

Main camera parameters

- Camera position
- Central direction
- World up direction
- Field of view & aspect ratio

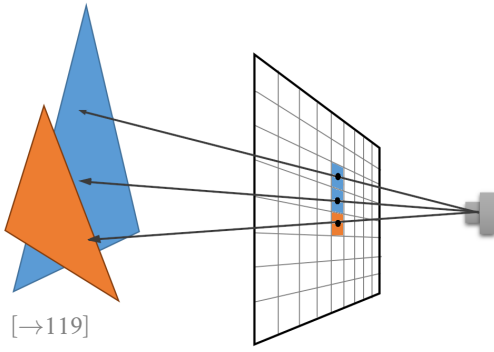
Ray Casting

The basic ray casting outline is

- Set up ray
  - Camera position is at $\boldsymbol{o}$
  - Connect the camera position with a point on a pixel
  - Normalize to get the direction to that point $\boldsymbol{d}$
  - Ray $r = \boldsymbol{o} + t\boldsymbol{d}$
- Find intersection with closest primitive
- Compute color for that intersection

To obtain $\boldsymbol{d}$ we have to find a pixel position in our 3D scene, i.e. we have to set up a mapping between our world and our target image.

To this end we imagine the image as a plane in front of the camera, one unit along its main viewing direction $\boldsymbol{e}$, such that the plane covers the area spanned by the field of view angles.



[→119]

Main camera parameters

- Camera position
- Central direction
- World up direction
- Field of view & aspect ratio

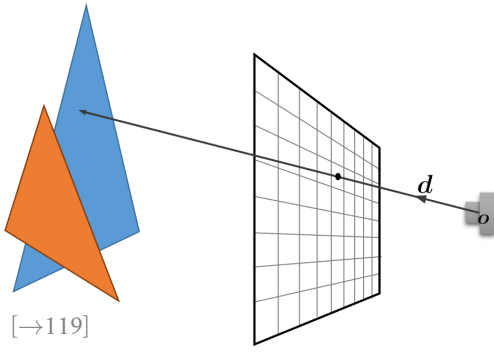
Ray Casting

The basic ray casting outline is

- Set up ray
  - Camera position is at $\boldsymbol{o}$
  - Connect the camera position with a point on a pixel
  - Normalize to get the direction to that point $\boldsymbol{d}$
  - Ray $r = \boldsymbol{o} + t\boldsymbol{d}$
- Find intersection with closest primitive
- Compute color for that intersection

To obtain $\boldsymbol{d}$ we have to find a pixel position in our 3D scene, i.e. we have to set up a mapping between our world and our target image.

To this end we imagine the image as a plane in front of the camera, one unit along its main viewing direction $\boldsymbol{e}$, such that the plane covers the area spanned by the field of view angles.



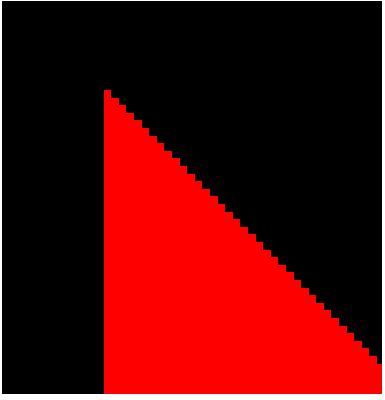
[→119]

Main camera parameters

- Camera position
- Central direction
- World up direction
- Field of view & aspect ratio

Sampling 101

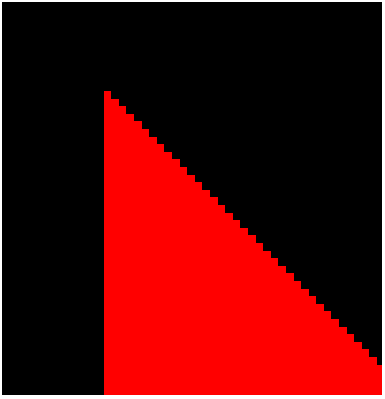
When we compute the colors for pixels we **sample** the scene for certain directions. The image is then a reconstruction of the scene, based on the sampled data. Uniformly sampling only the pixel centers is problematic: [→121]



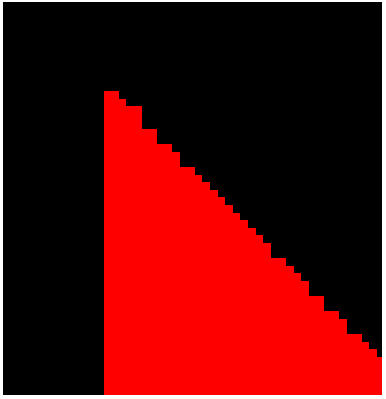
Sample center

Sampling 101

When we compute the colors for pixels we **sample** the scene for certain directions. The image is then a reconstruction of the scene, based on the sampled data. Uniformly sampling only the pixel centers is problematic: [→121]



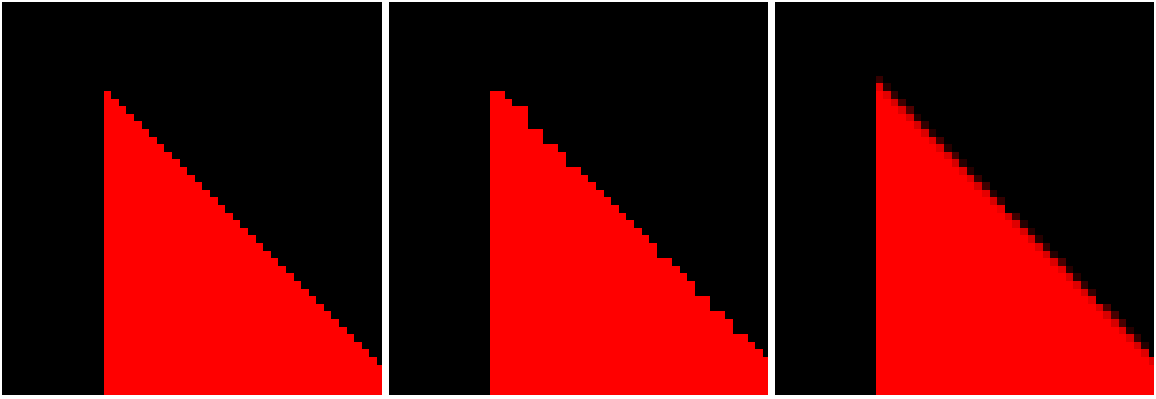
Sample center



Sample randomly in pixel

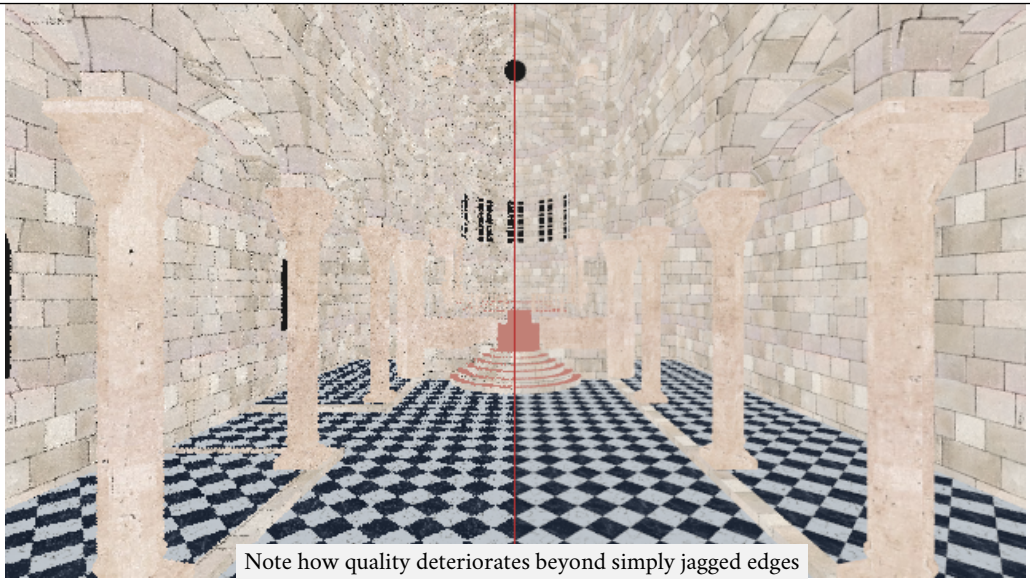
Sampling 101

When we compute the colors for pixels we **sample** the scene for certain directions. The image is then a reconstruction of the scene, based on the sampled data. Uniformly sampling only the pixel centers is problematic: [→121]



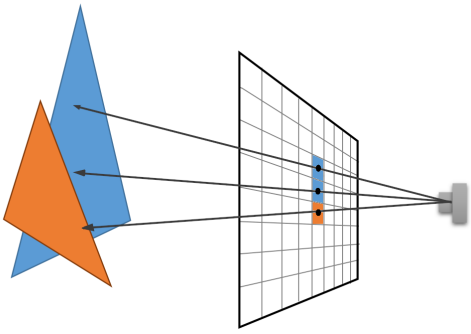
Sample center | Sample randomly in pixel | Integrate

Comparison — Sampling 101



1×

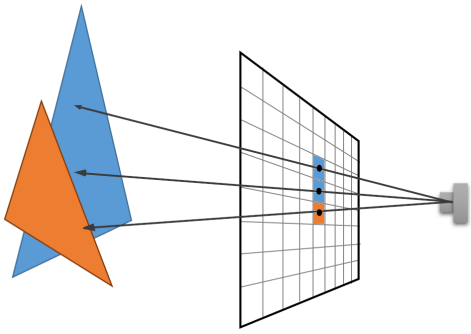
16×



Finding the closest primitive can be implemented very naively:

- Go over all primitives
- Compute the intersection of the ray with the primitive
- Store the closest one found

Note that this is unspeakably inefficient, but will do as a first stab in this course.

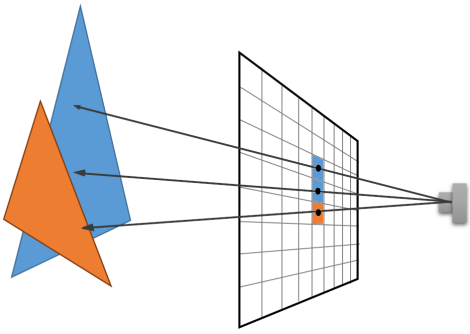


Finding the closest primitive can be implemented very naively:

- Go over all primitives
- Compute the intersection of the ray with the primitive
- Store the closest one found

Note that this is unspeakably inefficient, but will do as a first stab in this course.

This leaves the ray/primitive intersection as the last point before we can implement a basic ray casting algorithm.



Finding the closest primitive can be implemented very naively:

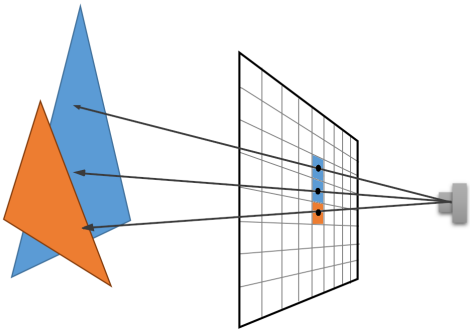
- Go over all primitives
- Compute the intersection of the ray with the primitive
- Store the closest one found

Note that this is unspeakably inefficient, but will do as a first stab in this course.

This leaves the ray/primitive intersection as the last point before we can implement a basic ray casting algorithm.

For triangles the outline is

- Find the intersection of the ray with the plane defined by the triangle
- Check if that intersection lies inside the triangle
- [→122]



Finding the closest primitive can be implemented very naively:

- Go over all primitives
- Compute the intersection of the ray with the primitive
- Store the closest one found

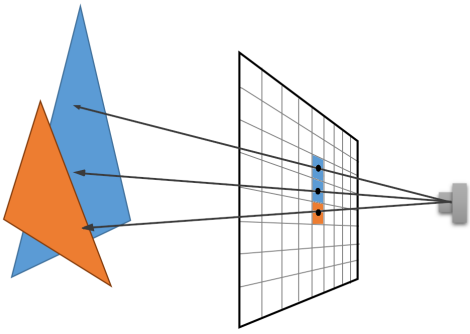
Note that this is unspeakably inefficient, but will do as a first stab in this course.

This leaves the ray/primitive intersection as the last point before we can implement a basic ray casting algorithm.

For triangles the outline is

- Find the intersection of the ray with the plane defined by the triangle
- Check if that intersection lies inside the triangle
- [→122] $\boldsymbol{o} + t\boldsymbol{d} = a + \beta(b-a) + \gamma(c-a)$ yields (β, γ, t).
- (β, γ) and t can both be used to find intersection point, t gives distance to camera, (β, γ) interpolate vertex attributes

Ray/Primitive Intersection



The computation exhibits many common terms, see Shirley's book (excerpt on GRIPS).

Finding the closest primitive can be implemented very naively:

- Go over all primitives
- Compute the intersection of the ray with the primitive
- Store the closest one found

Note that this is unspeakably inefficient, but will do as a first stab in this course.

This leaves the ray/primitive intersection as the last point before we can implement a basic ray casting algorithm.

For triangles the outline is

- Find the intersection of the ray with the plane defined by the triangle
- Check if that intersection lies inside the triangle
- [→122] $\boldsymbol{o} + t\boldsymbol{d} = a + \beta(b-a) + \gamma(c-a)$ yields (β, γ, t).
- (β, γ) and t can both be used to find intersection point, t gives distance to camera, (β, γ) interpolate vertex attributes

With all that, we can formulate a simple ray tracer:

```
camera cam
for ((x, y) in image)
    ray r = cam(x, y)
    (t, β, γ, Δ) = (∞, 0, 0, ∅)
```

With all that, we can formulate a simple ray tracer:

```
camera cam
for ((x, y) in image)
    ray r = cam(x, y)
    (t, β, γ, Δ) = (∞, 0, 0, ∅)
    for (Δ_i in list_of_triangles)
        (hit, t_i, β_i, γ_i) = intersect(r, Δ_i)
```

With all that, we can formulate a simple ray tracer:

```
camera cam
for ((x, y) in image)
    ray r = cam(x, y)
    (t, β, γ, Δ) = (∞, 0, 0, ∅)
    for (Δ_i in list_of_triangles)
        (hit, t_i, β_i, γ_i) = intersect(r, Δ_i)
        if (hit && t_i < t)
            (t, β, γ, Δ) = (t_i, β_i, γ_i, Δ_i)
```

With all that, we can formulate a simple ray tracer:

```
camera cam
for ((x, y) in image)
    ray r = cam(x, y)
    (t, β, γ, Δ) = (∞, 0, 0, ∅)
    for (Δ_i in list_of_triangles)
        (hit, t_i, β_i, γ_i) = intersect(r, Δ_i)
        if (hit && t_i < t)
            (t, β, γ, Δ) = (t_i, β_i, γ_i, Δ_i)
    if (t ≠ ∞)
        return color(Δ, β, γ)
```

With all that, we can formulate a simple ray tracer:

```
camera cam
for ((x, y) in image)
    ray r = cam(x, y)
    (t, β, γ, Δ) = (∞, 0, 0, ∅)
    for (Δ_i in list_of_triangles)
        (hit, t_i, β_i, γ_i) = intersect(r, Δ_i)
        if (hit && t_i < t)
            (t, β, γ, Δ) = (t_i, β_i, γ_i, Δ_i)
    if (t ≠ ∞)
        return color(Δ, β, γ)
    else
        return background(x, y)
```

With all that, we can formulate a simple ray tracer:

```
camera cam
for ((x, y) in image)
    ray r = cam(x, y)
    (t, β, γ, Δ) = (∞, 0, 0, ∅)
    for (Δ_i in list_of_triangles)
        (hit, t_i, β_i, γ_i) = intersect(r, Δ_i)
        if (hit && t_i < t)
            (t, β, γ, Δ) = (t_i, β_i, γ_i, Δ_i)
    if (t ≠ ∞)
        return color(Δ, β, γ)
    else
        return background(x, y)
```

This is too simplistic as the search for the closest intersection in n triangles is $O(n)$ (actually, precisely n) and because this linear search is executed for each pixel, thus the complexity is $O(n \cdot w \cdot h)$.

With all that, we can formulate a simple ray tracer:

```
camera cam
for ((x, y) in image)
    ray r = cam(x, y)
    (t, β, γ, Δ) = (∞, 0, 0, ∅)
    for (Δ_i in list_of_triangles)
        (hit, t_i, β_i, γ_i) = intersect(r, Δ_i)
        if (hit && t_i < t)
            (t, β, γ, Δ) = (t_i, β_i, γ_i, Δ_i)
    if (t ≠ ∞)
        return color(Δ, β, γ)
    else
        return background(x, y)
```

This is too simplistic as the search for the closest intersection in n triangles is $O(n)$ (actually, precisely n) and because this linear search is executed for each pixel, thus the complexity is $O(n \cdot w \cdot h)$.

Rendering a scene with $n = 12{,}000{,}000$ and $w = 3840, h = 2160$, for a single sample this would entail $99{,}532{,}800{,}000{,}000$ (about 100 trillion) ray/primitive intersections.

With all that, we can formulate a simple ray tracer:

```
camera cam
for ((x, y) in image)
    ray r = cam(x, y)
    (t, β, γ, Δ) = (∞, 0, 0, ∅)
    for (Δ_i in list_of_triangles)
        (hit, t_i, β_i, γ_i) = intersect(r, Δ_i)
        if (hit && t_i < t)
            (t, β, γ, Δ) = (t_i, β_i, γ_i, Δ_i)
    if (t ≠ ∞)
        return color(Δ, β, γ)
    else
        return background(x, y)
```

This is too simplistic as the search for the closest intersection in n triangles is $O(n)$ (actually, precisely n) and because this linear search is executed for each pixel, thus the complexity is $O(n \cdot w \cdot h)$.

Rendering a scene with $n = 12{,}000{,}000$ and $w = 3840, h = 2160$, for a single sample this would entail $99{,}532{,}800{,}000{,}000$ (about 100 trillion) ray/primitive intersections.

A single ray/tri intersection is around 40 Flops, i.e. the example would require around 4 quadrillion floating point operations, taking around two minutes on a current generation top-of-the-line GPU (if it could be fully utilized)

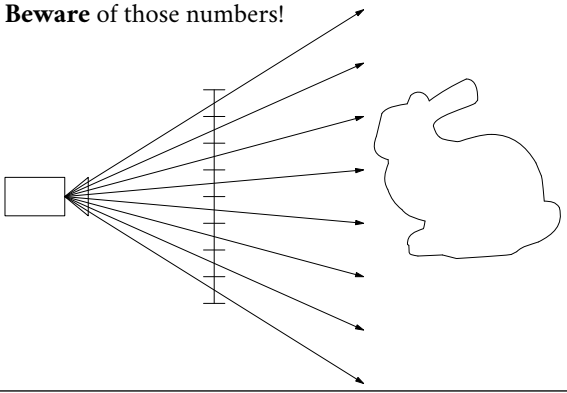
…and the image would look horrible ;)

Finding the closest Ray/Triangle intersection is a linear search-problem, i.e. $O(n)$. With proper metrics such problems can be improved by searching in a tree instead, i.e. $O(\log n)$.

Applied to our earlier example using $n = 12{,}000{,}000$, $w = 3840, h = 2160$ our computation times would go down to 0.2 milliseconds!
Beware of those numbers!

Finding the closest Ray/Triangle intersection is a linear search-problem, i.e. $O(n)$. With proper metrics such problems can be improved by searching in a tree instead, i.e. $O(\log n)$.

Applied to our earlier example using $n = 12{,}000{,}000$, $w = 3840, h = 2160$ our computation times would go down to 0.2 milliseconds!
Beware of those numbers!

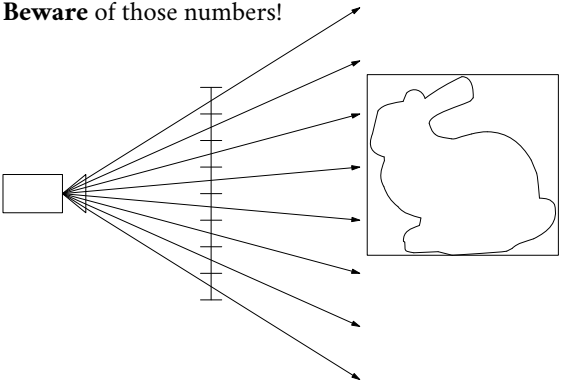


Hierarchy of bounding volumes (here bunny model of around 70,000 triangles

- Put small sets of primitives into a box
- Put smaller boxes into bigger boxes
- During tracing: recursively test boxes
- Skips a tremendous amount of geometry

Finding the closest Ray/Triangle intersection is a linear search-problem, i.e. $O(n)$. With proper metrics such problems can be improved by searching in a tree instead, i.e. $O(\log n)$.

Applied to our earlier example using $n = 12{,}000{,}000$, $w = 3840, h = 2160$ our computation times would go down to 0.2 milliseconds!
Beware of those numbers!

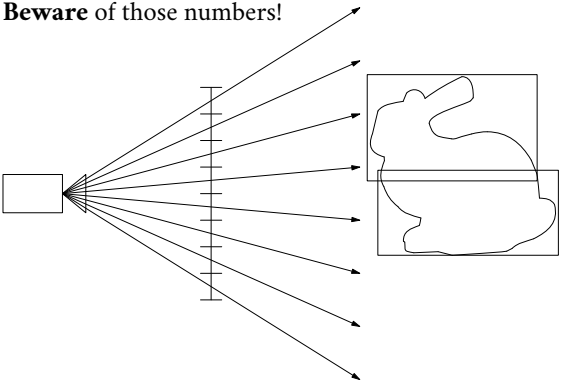


Hierarchy of bounding volumes (here bunny model of around 70,000 triangles

- Put small sets of primitives into a box
- Put smaller boxes into bigger boxes
- During tracing: recursively test boxes
- Skips a tremendous amount of geometry

Finding the closest Ray/Triangle intersection is a linear search-problem, i.e. $O(n)$. With proper metrics such problems can be improved by searching in a tree instead, i.e. $O(\log n)$.

Applied to our earlier example using $n = 12{,}000{,}000$, $w = 3840, h = 2160$ our computation times would go down to 0.2 milliseconds!
Beware of those numbers!

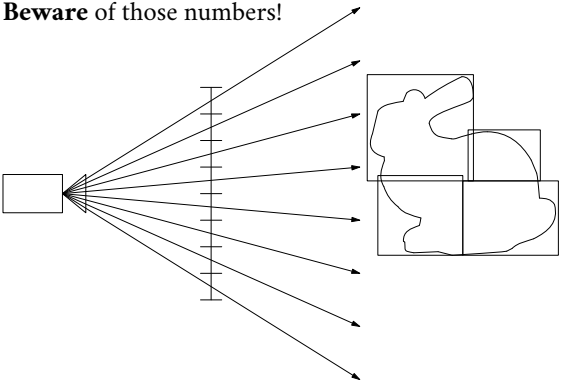


Hierarchy of bounding volumes (here bunny model of around 70,000 triangles

- Put small sets of primitives into a box
- Put smaller boxes into bigger boxes
- During tracing: recursively test boxes
- Skips a tremendous amount of geometry

Finding the closest Ray/Triangle intersection is a linear search-problem, i.e. $O(n)$. With proper metrics such problems can be improved by searching in a tree instead, i.e. $O(\log n)$.

Applied to our earlier example using $n = 12{,}000{,}000$, $w = 3840, h = 2160$ our computation times would go down to 0.2 milliseconds!
Beware of those numbers!

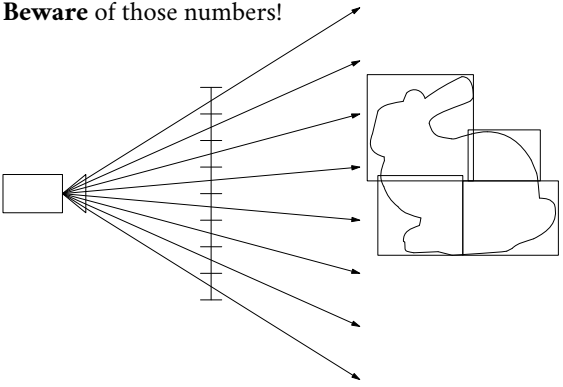


Hierarchy of bounding volumes (here bunny model of around 70,000 triangles

- Put small sets of primitives into a box
- Put smaller boxes into bigger boxes
- During tracing: recursively test boxes
- Skips a tremendous amount of geometry

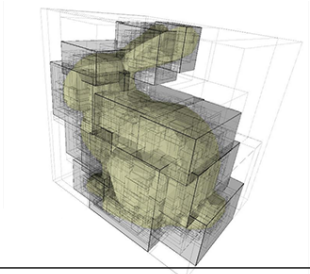
Finding the closest Ray/Triangle intersection is a linear search-problem, i.e. $O(n)$. With proper metrics such problems can be improved by searching in a tree instead, i.e. $O(\log n)$.

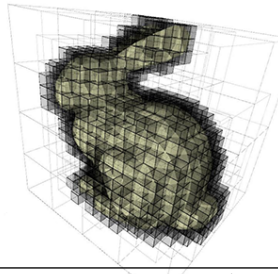
Applied to our earlier example using $n = 12{,}000{,}000$, $w = 3840, h = 2160$ our computation times would go down to 0.2 milliseconds!
Beware of those numbers!



Hierarchy of bounding volumes (here bunny model of around 70,000 triangles

- Put small sets of primitives into a box
- Put smaller boxes into bigger boxes
- During tracing: recursively test boxes
- Skips a tremendous amount of geometry





The most commonly used structure (in the last decade) is the Bounding Volume Hierarchy (BVH), as roughly described on the previous slide. Other approaches (that we will not further discuss here) are kd-trees, octrees, grids, intervals, etc.

The most common choice of bounding volume (proxy geometry) is axis aligned bounding boxes (AABB). [→others]

The construction of such a BVH can be done in different ways, we will focus on the most established ones using a top-down, median cut approach. The general layout is as follows:

```
node_ref subdivide(triangle *tris, int s, int e) {
    if (e-s < N_leaf)
        return make_leaf(tris, s, e);
    aabb box = aabb(tris, s, e);
    node_ref node = make_internal(box);
    int mid = split(tris, s, e);
    node.left  = subdivide(tris, s, mid);
    node.right = subdivide(tris, mid, e);
    return node;
}
```

There are different variations of the `split` function:

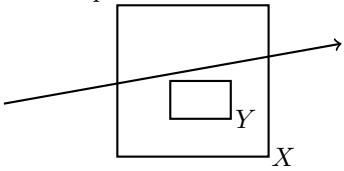
- **Object Median**
  - Sort the primitives along the position on the longest axis of their bounding box
  - Choose the split position such that the number of primitives on either side is equal
  - Note: how to compare primitives by position? Approximate by using center of gravity
  - [→draw, show problems]

There are different variations of the `split` function:

- **Object Median**
  - Sort the primitives along the position on the longest axis of their bounding box
  - Choose the split position such that the number of primitives on either side is equal
  - Note: how to compare primitives by position? Approximate by using center of gravity
  - [→draw, show problems]
- **Spatial Median**
  - Select split position on the spatial center of bounding box with respect to longest axis
  - Partition the primitives wrt. this position into 'left' and 'right'
  - [→draw, show corner case]

There are different variations of the `split` function:

- **Object Median**
  - Sort the primitives along the position on the longest axis of their bounding box
  - Choose the split position such that the number of primitives on either side is equal
  - Note: how to compare primitives by position? Approximate by using center of gravity
  - [→draw, show problems]
- **Spatial Median**
  - Select split position on the spatial center of bounding box with respect to longest axis
  - Partition the primitives wrt. this position into 'left' and 'right'
  - [→draw, show corner case]
- **Surface Area Heuristic**
  - Neither OM nor SM are ideal
  - Basic idea: Probability that ray intersects an enclosed bounding volume corresponds to relative surface areas of parent and child box.



$P(Y|X) \approx A(Y)/A(X)$

  - Use to minimize cost of a split position s that partitions a box P into L and R, each holding $n(L)$ and $n(R)$ primitives:
    $c(L, R, P) = P(L|P)n(L) + P(R|P)n(R)$
    → Find $\arg\min_{L,R} c(L, R, P)$ [→expl 2021/52]

Ray/Box Intersection

We want to use bounding volumes (in our case AABBs) to determine if a ray could intersect a part of the scene that is bounded by this volume. To this end we have to compute the ray/box intersection.

Terminology:

- A box b is a pair of the minimal point and the maximal point of the box for each axis
$$(\boldsymbol{b}^-, \boldsymbol{b}^+) = \left(\begin{pmatrix} b_x^- \\ b_y^- \\ b_z^- \end{pmatrix}, \begin{pmatrix} b_x^+ \\ b_y^+ \\ b_z^+ \end{pmatrix} \right)$$

Ray/Box Intersection

We want to use bounding volumes (in our case AABBs) to determine if a ray could intersect a part of the scene that is bounded by this volume. To this end we have to compute the ray/box intersection.

Terminology:

- A box b is a pair of the minimal point and the maximal point of the box for each axis $(\boldsymbol{b}^-, \boldsymbol{b}^+) = \left(\begin{pmatrix} b_x^- \\ b_y^- \\ b_z^- \end{pmatrix}, \begin{pmatrix} b_x^+ \\ b_y^+ \\ b_z^+ \end{pmatrix} \right)$
- We sometimes write variables in vector indices to select specific vector components, e.g. with $v = (123)^\mathsf{T}$ we say v_c to select any of the three components, depending on what value c holds , i.e. $\sum_{c \in \{x,y,z\}} v_c = v_x + v_y + v_z$.

We want to use bounding volumes (in our case AABBs) to determine if a ray could intersect a part of the scene that is bounded by this volume. To this end we have to compute the ray/box intersection.

Terminology:

- A box b is a pair of the minimal point and the maximal point of the box for each axis $(\boldsymbol{b}^-, \boldsymbol{b}^+) = \left(\begin{pmatrix} b_x^- \\ b_y^- \\ b_z^- \end{pmatrix}, \begin{pmatrix} b_x^+ \\ b_y^+ \\ b_z^+ \end{pmatrix} \right)$
- We sometimes write variables in vector indices to select specific vector components, e.g. with $v = (123)^\mathsf{T}$ we say v_c to select any of the three components, depending on what value c holds , i.e. $\sum_{c \in \{x,y,z\}} v_c = v_x + v_y + v_z$.
- If a ray overlaps a box this overlap is an interval $t_b = [t_b^-, t_b^+]$ in the parameter space
- [→2021/46]

Ray/Box Intersection

We want to use bounding volumes (in our case AABBs) to determine if a ray could intersect a part of the scene that is bounded by this volume. To this end we have to compute the ray/box intersection.

Terminology:

- A box b is a pair of the minimal point and the maximal point of the box for each axis $(\boldsymbol{b}^-, \boldsymbol{b}^+) = \left(\begin{pmatrix} b_x^- \\ b_y^- \\ b_z^- \end{pmatrix}, \begin{pmatrix} b_x^+ \\ b_y^+ \\ b_z^+ \end{pmatrix} \right)$
- We sometimes write variables in vector indices to select specific vector components, e.g. with $v = (123)^\mathsf{T}$ we say v_c to select any of the three components, depending on what value c holds , i.e. $\sum_{c \in \{x,y,z\}} v_c = v_x + v_y + v_z$.
- If a ray overlaps a box this overlap is an interval $t_b = [t_b^-, t_b^+]$ in the parameter space
- [→2021/46]

Summary: A ray $\boldsymbol{o} + t\boldsymbol{d}$ intersects a box $(\boldsymbol{b}^-, \boldsymbol{b}^+)$ if $t_b = t_x \cap t_y \cap t_z \neq \emptyset \wedge t_b^- \geq 0$, where

Ray/Box Intersection

We want to use bounding volumes (in our case AABBs) to determine if a ray could intersect a part of the scene that is bounded by this volume. To this end we have to compute the ray/box intersection.

Terminology:

- A box b is a pair of the minimal point and the maximal point of the box for each axis $(\boldsymbol{b}^-, \boldsymbol{b}^+) = \left(\begin{pmatrix} b_x^- \\ b_y^- \\ b_z^- \end{pmatrix}, \begin{pmatrix} b_x^+ \\ b_y^+ \\ b_z^+ \end{pmatrix} \right)$
- We sometimes write variables in vector indices to select specific vector components, e.g. with $v = (123)^\mathsf{T}$ we say v_c to select any of the three components, depending on what value c holds , i.e. $\sum_{c \in \{x,y,z\}} v_c = v_x + v_y + v_z$.
- If a ray overlaps a box this overlap is an interval $t_b = [t_b^-, t_b^+]$ in the parameter space
- [→2021/46]

Summary: A ray $\boldsymbol{o} + t\boldsymbol{d}$ intersects a box $(\boldsymbol{b}^-, \boldsymbol{b}^+)$ if $t_b = t_x \cap t_y \cap t_z \neq \emptyset \wedge t_b^- \geq 0$, where

$$t_c^{+/-} = \begin{cases} \frac{b_c^{+/-} - o_c}{d_c} & d_c \neq 0 \\ +/- \infty & d_c = 0 \wedge o_c \in [b_c^-, b_c^+] \\ \emptyset & \text{otherwise} \end{cases}$$

With all that, we can formulate a much more efficient ray tracer:

```
camera cam
for ((x, y) in image)
  ray r = cam(x, y)
  (t, β, γ, Δ) = bvh.intersect(r)
  if (t ≠ ∞)
    return color(Δ, β, γ)
  else
    return background(x, y)
```

With all that, we can formulate a much more efficient ray tracer:

```
camera cam
for ((x, y) in image)
  ray r = cam(x, y)
  (t, β, γ, Δ) = bvh.intersect(r)
  if (t ≠ ∞)
    return color(Δ, β, γ)
  else
    return background(x, y)
```

More interesting is then of course how the closest hitpoint is found via the BVH:

```
is_res bvh::intersect(ray r, node *n = root)
  if (n->overlaps(ray))
    if (n->is_leaf())
      return intersect(r,n->Δ)
    (t_L, β_L, γ_L, Δ_L) = bvh.intersect(r, n->left)
    (t_R, β_R, γ_R, Δ_R) = bvh.intersect(r, n->right)
    if (t_L < t_R)
      return (t_L, β_L, γ_L, Δ_L)
    else
      return (t_R, β_R, γ_R, Δ_R)
  else
    return (∞, 0, 0, ∅)
```

Note: This code is conceptual, the implementation is a little more involved as this is usually heavily optimized.

```
:wq
```