VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Computer Graphics I

460-4078

Fall 2024

Last update 24. 10. 2024

Acceleration Data Structures

- Ray-primitive intersection phase is the most demanding part of every ray tracer
- Naive implementation of a ray tracing algorithm for a single ray has $O(n)$ time complexity, where n is number of primitives (e.g. triangles) in the scene
- Spatial-hierarchy data structures reduce the search space exponentially (with the time complexity $O(\log n)$)
- Time complexity required to build the structure is in average case in $O(n \log n)$ and $O(n^2)$ in worst case (sorted triangles)
- There are basicaly two approaches
  - Object subdivision – grouping of nearby triangles (e.g. BVH, BIH)
  - Space subdivision – divides space into sub-spaces with assigned triangles contained in these spaces (e.g. kD-tree, BSP, Octree, Grid)

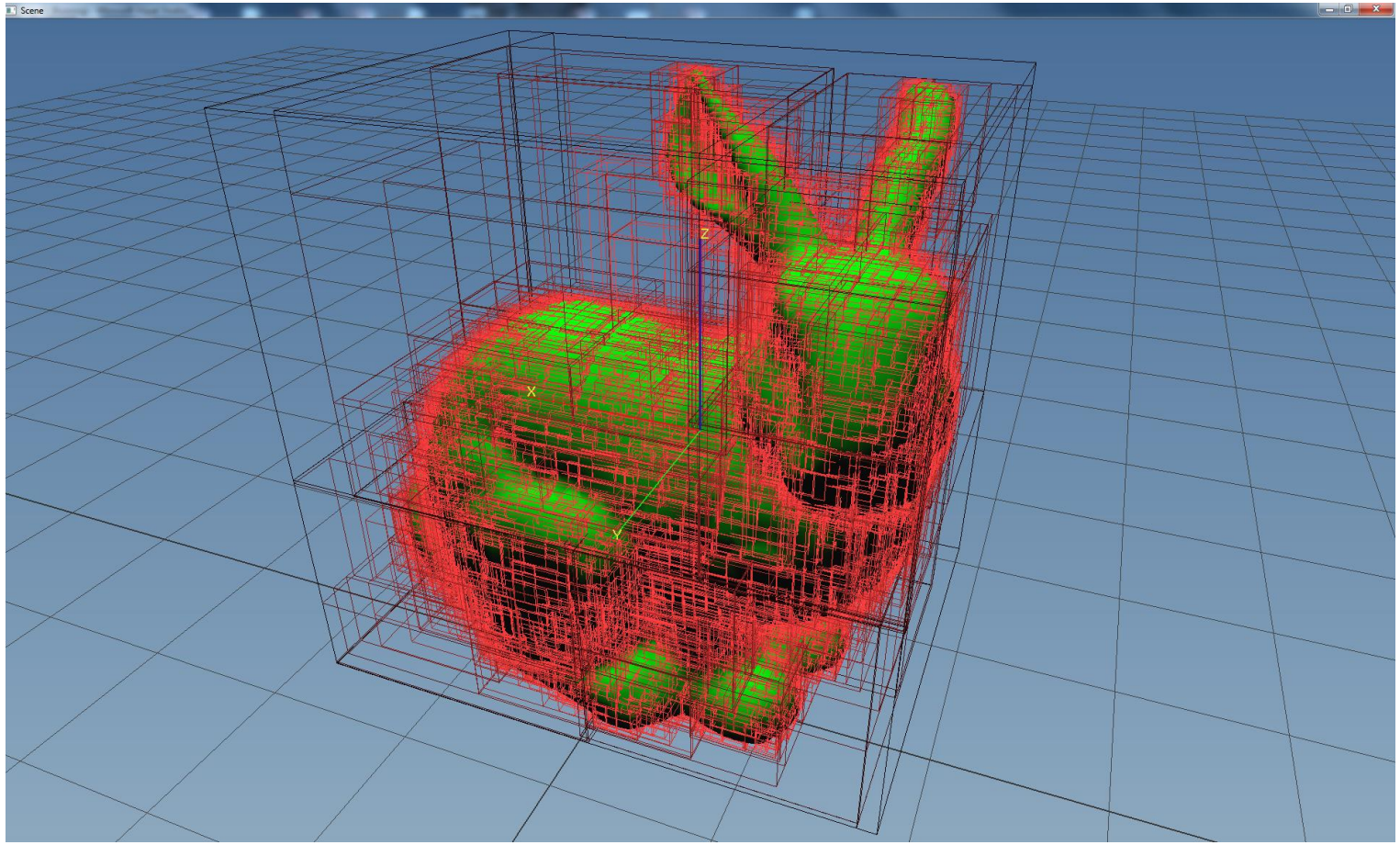
Acceleration Data Structures

- The most commonly used structures
  - Octrees
  - **Boundary-Volume Hierarchies (e.g. BVH)**
  - kD-trees
- These algorithms follow divide and conquer scheme and are relatively straightforward in implementation
- The biggest issue is to determine the best split position of a node on each level of three hierarchy. In other words, we don't know which objects should be grouped together or where to cut space into two subsequent nodes to achieve best performance during three traversal
- We may assume that creating an optimal accelerator is an NP-hard problem

Bounding Volume Hierarchy (BVH)

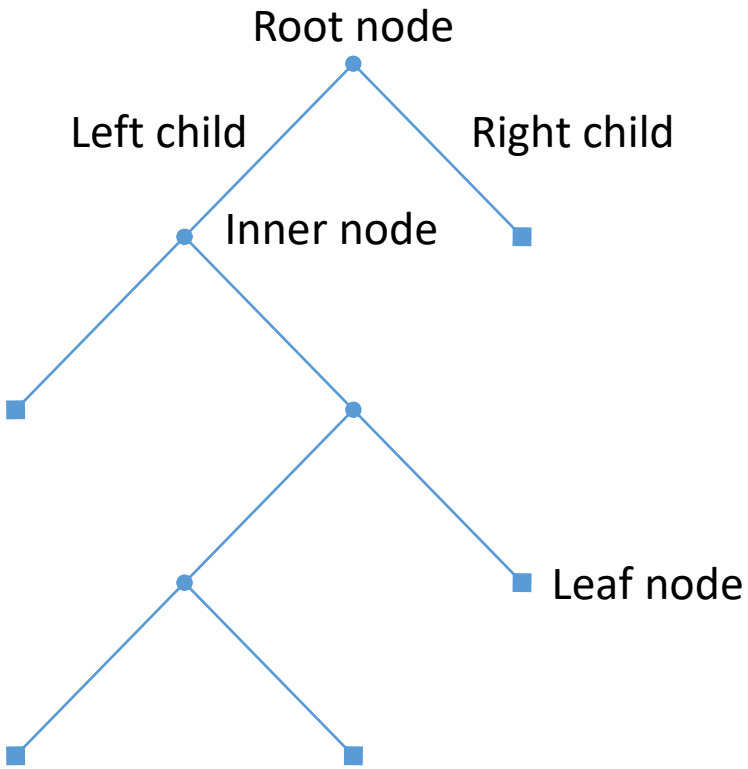
- Used to reduce the number of ray-primitive intersection tests
- A tree structure on a set of geometric objects (triangles in our case)
- In the top-down approach, we start with the entire scene enclosed in a one large bounding volume – a root node
- Bounding volumes are typically represented as an axis aligned bounding box (AABB)
- We recursively divide nodes until each node consists of only a single primitive (or a small number of primitives) thus obtaining a tree structure
- Ideal BVH minimizes number of ray-primitive and ray-AABB intersection tests

Bounding Volume Hierarchy (BVH)



Tree

- Each node contains a list of included triangles and geometry parameters of the (axis aligned) bounding box (compact node has 32 bytes in size)



Bounding Volume Hierarchy (BVH)

```
struct Node
{
  AABB bounds;
  int span[2];
  Node * children[2]; // idx 0 is left, 1 is right

  Node( int from, int to )
  {
    span[0] = from;
    span[1] = to;
    children[0] = children[1] = nullptr;
  }

  ~Node()
  {
    delete children[1];
    delete children[0];
  }
};
```

```
class BVH
{
public:
  BVH( std::vector<Triangle *> * items );
  ~BVH();

  void BuildTree();
  void Traverse( Ray & ray );

private:
  Node * BuildTree( int from, int to, int depth );
  void Traverse( Ray & ray, Node * node, int depth );
  AABB GetNodeAABB( from, to );

  Node * root_;
  std::vector<Triangle *> * items_;
};
```

Build Tree

- Sequence of steps in BuildTree method:
  - Create a new node containing items in range $\langle from, to \rangle$ (in case of the root node, it is $\langle 0, n-1 \rangle$)
  - Set the AABB of all items in the current node
  - If the number of items in the current node is greater then e.g. 4 proceed with node splitting:
    - Choose a split axis (the simplest scheme is to repeatedly alternate axis with each level of the tree as follows 0:x, 1:y, 2:z, 3:x, 4:y, 5:z, 6:x,…)
    - Choose a $pivot$ (e.g. index of the middle item from the current range)
    - Sort items from the current range along the chosen split axis
    - Split the current node into a left child $\langle from, pivot-1 \rangle$ and a right child $\langle pivot, to \rangle$
  - Return the current node

Build Tree

```
Node * BVH::BuildTree( int from, int to, int depth )
{
  Node * node = new Node( from, to );
  node->bounds = GetNodeAABB( from, to ); // get the bounds of all vertices of all
triangles in the current range
  if ( to - from + 1 > max_leaf_items ) // e.g. max_leaf_items = 4
  {
    int split_axis = depth % 3; // alt. the longest axis of the current AABB
    int pivot = ( to - from + 1 ) / 2 + from; // e.g. the mid index
    // sort triangles here – see “How to sort triangles“ slide
    node->children[0] = BuildTree( from, pivot - 1, depth + 1 );
    node->children[1] = BuildTree( pivot, to, depth + 1 );
  }

  return node;
}
```

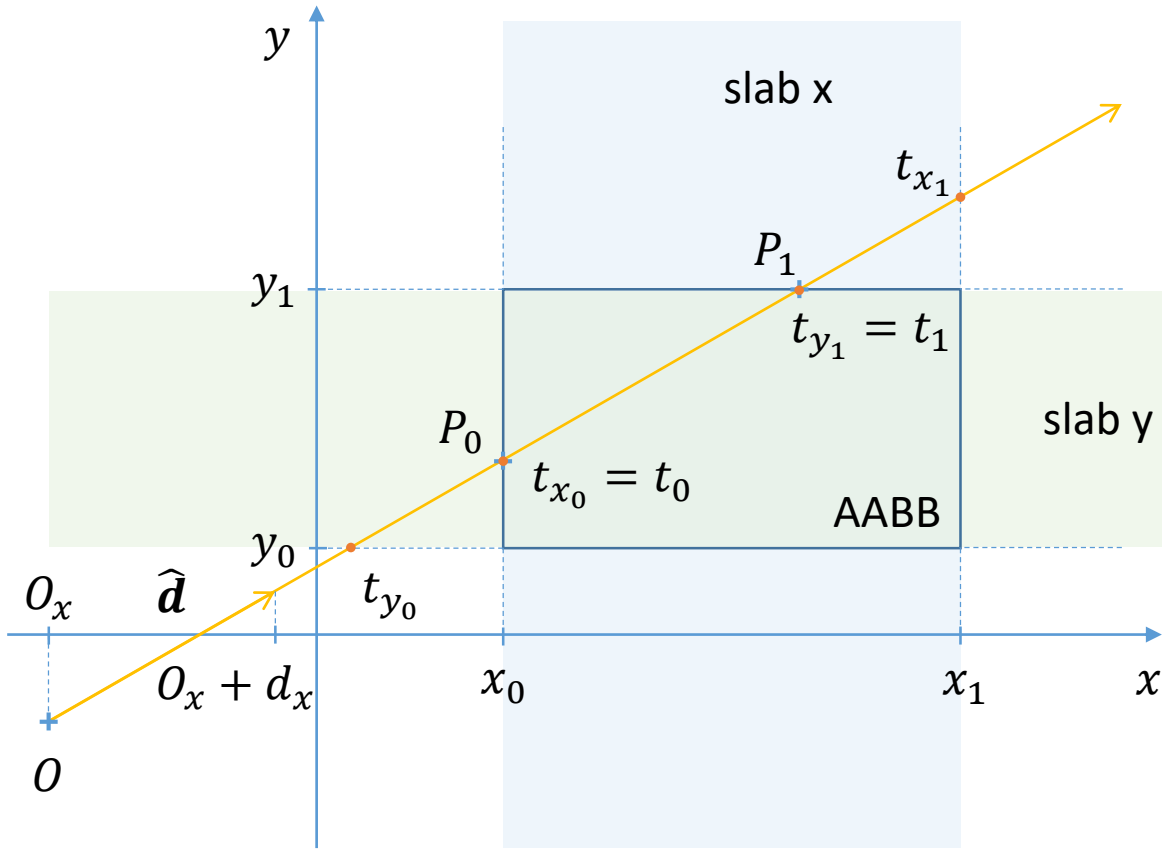
Traverse Tree

- Sequence of steps in Traverse method:
  - If ray vs current node bounds intersection exists:
    - If the current node is a leaf
      - Update the ray.tfar parameter based on possible intersections with triangles in the current node
    - Otherwise traverse left (first) and right (second) child subsequently

Traverse Tree

```
void BVH::Traverse( Ray & ray, Node * node, float t0, float t1, int depth )
{
  if ( RayBoxIntersection( ray, node->bounds, t0, t1 ) ) {
    if ( node is leaf ) {
      for ( int i = node->span[0]; i <= node->span[1]; ++i ) {
        if ( RayTriangleIntersection( ray, items_[i] ) ) {
          // we assume that the ray.tfar is updated in the intersection method
          t1 = ray.tfar;
        }
      }
    } else { // node is inner node
      Traverse( ray, node->children[0], t0, ray.tfar, depth + 1 );
      Traverse( ray, node->children[1], t0, ray.tfar, depth + 1 );
    }
  } else {
    // ray misses this node (and also its children) – nothing to do here
  }
}
```

Slab Test (Ray vs AABB Intersection)



$r(t) = O + \hat{\boldsymbol{d}}t$

$t_{x_0} = \frac{x_0 - O_x}{d_x}, t_{y_0} = \ldots$

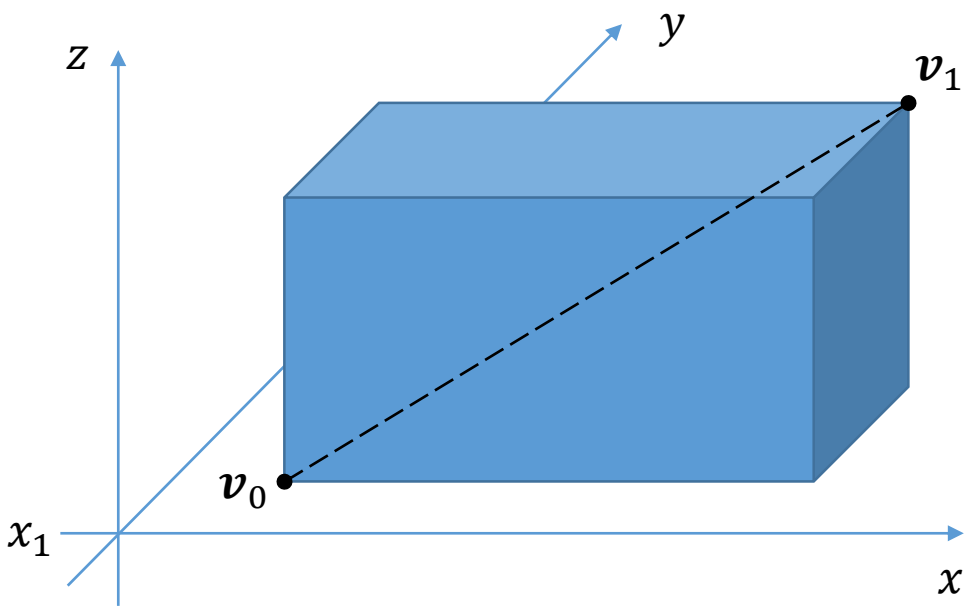
Note that the semantics of near and far intersections must be persevered regardless the ray direction, i.e.
if $t_{x_0} > t_{x_1}$ then swap(t_{x_0}, t_{x_1})
The same holds for y and z axis

$t_0 = \max(t_{x_0}, t_{y_0}, t_{z_0})$
$t_1 = \min(t_{x_1}, t_{y_1}, t_{z_1})$

if $t_0 < t_1$ and $0 < t_1$ then $P_{\{0,1\}} = r(t_{\{0,1\}})$
else miss

Axis Aligned Bounding Box (AABB)

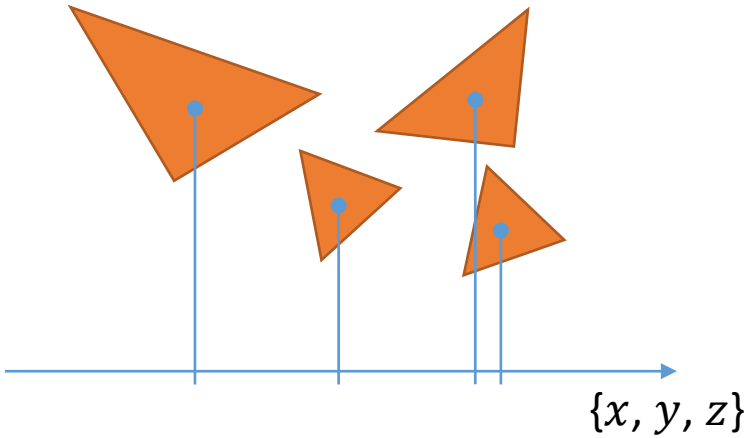
```
struct AABB
{
  Vector3f bounds[2];

  AABB( Vector3f v0, Vector3f v1 )
  {
    bounds[0] = v0;
    bounds[1] = v1;
  }

  float SurfaceArea() // for SAH
};
```



How to Sort Triangles

- We can use barycenters to sort triangles along a given axis
- Then it's the same problem as sorting real numbers

```
struct TriangleComparator {
  TriangleComparator( const char axis ) { axis_ = axis; }
  bool operator() ( Triangle * a, Triangle * b ) {
    return a->center().data[axis_] < b->center().data[axis_];
  }
  char axis_;
};
…
std::vector<Triangle *>::iterator begin = items_->begin();
std::nth_element( begin + from, begin + pivot, begin + to + 1,  TriangleComparator( split_axis ) ); // O(n) in avg
```



Making BVH Faster

- Store the inverse ray direction with the ray
  - This will accelerate ray-bounding box test
- Do an early out in ray traversal
  - Do not traverse any further when you already know that the ray can not hit anything else
- Build the tree using the Surface Area Heuristic (SAH) (2× speedup)
  - Greedy algorithm that minimizes the sum of the areas of the bounding boxes of children nodes in the given level of the tree. Binning yields nearly identical trees at a fraction of time. Also applicable in kD-trees
- Ray packet traversal
  - Traversing packets of coherent rays through a structure (SIMD vectorization)

Front-to-back (Ordered) Traversal

- Option 1: Calculate the minimum distance of both child nodes from ray origin and traverse the nearest child node first
- Option 2: Use ray direction sign for split axis to determine first and second node

```
first = children_[0];
second = children_[1];
if ( sgn( ray.dir[split_axis_] ) < 0 ) swap( first, second );
```

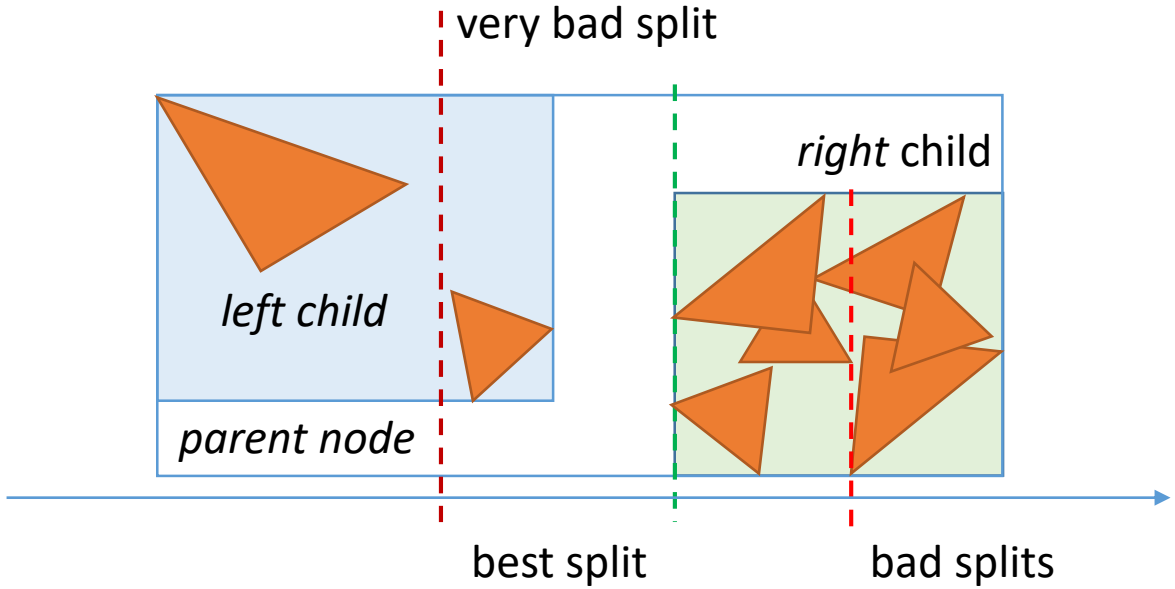
- Option 3: Same as option 2 but the split axis is determined by the axis for which the child node centroids are furthest apart

Front-to-back (Ordered) Traversal

- Note the following: nodes may overlap and a closer intersection may exist in further node
- But one thing is certain: nodes beyond an already found intersection may be skipped safely

Surface Area Heuristic (SAH)

- We assume uniform distribution of rays in our scene
- SAH improves ray tracer performance by preferring small *pricey* nodes with many triangles and large *cheap* nodes with a lot of empty space



We have a bigger chance to hit the left child than the right one consequently filtering out a larger portion of triangles when we choose the green split over the red ones

Surface Area Heuristic (SAH)

- The (minimized) cost of splitting a node into volumes A and B is defined as

$$C(A,B) = t_{tr} + p(A)\sum_{i=1}^{N_A} t_{int}(a_i) + p(B)\sum_{i=1}^{N_B} t_{int}(b_i)$$

t_{tr} … the time to traverse an interior node (constant, e.g. 1)

t_{int} … the time of one ray-triangle intersection (constant, e.g. 2)

$p(A),\ p(B)$ … probabilities that the ray passes through A, resp. B

$N_A,\ N_B$ … the number of triangles in volume A, resp. B

a_i, b_i … triangle in volume A, resp. B

Surface Area Heuristic (SAH)

Cauchy-Crofton formula yields the conditional probability: given two nested, convex, closed surfaces A and P, with A nested inside P, the probability of a random line intersecting the inner surface A, conditional on it intersecting the outer surface P, is as follows…

- We can compute the probabilities $p(A)$ as

$$p(A|P) = \frac{SA(A)}{SA(P)}$$

provided that A (left or right child) is a convex volume in another convex volume P (parent node) and SA is a surface area of given volume (e.g. area of a bounding box)

- We can simplify the previous equation as follows

$$C(A,B) = \frac{SA(A)}{SA(P)} N_A + \frac{SA(B)}{SA(P)} N_B$$

Super Sampling

Source: Mitchell, Don P. Consequences of stratified sampling in graphics. ACM, 1996.

- True pixel value x is estimated by the mean value $\bar{x}$ of N samples x_i from a small square area in an image

$$x \approx \bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

- The variance of M independent trials $\overline{x_j}$ is a measure of the accuracy of this estimate

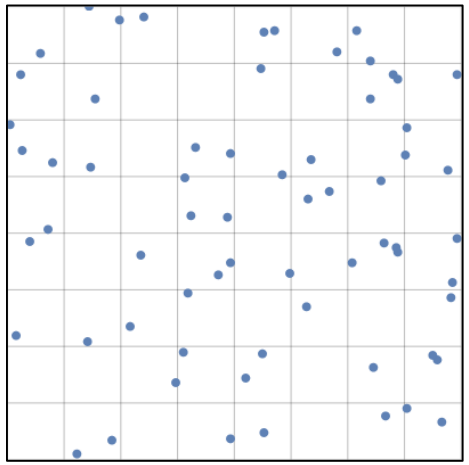
$$\sigma_x^2 = \frac{1}{M}\sum_{j=1}^{M}\left(\overline{x_j} - x\right)^2$$

- If the pixel area is sampled at uniformly distributed random locations, the central limit theorem implies that the variance of the mean is $O(1/N)$

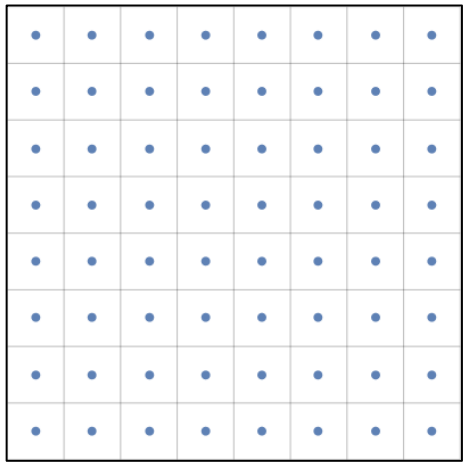
Super Sampling

Source: Mitchell, Don P. Consequences of stratified sampling in graphics. ACM, 1996.

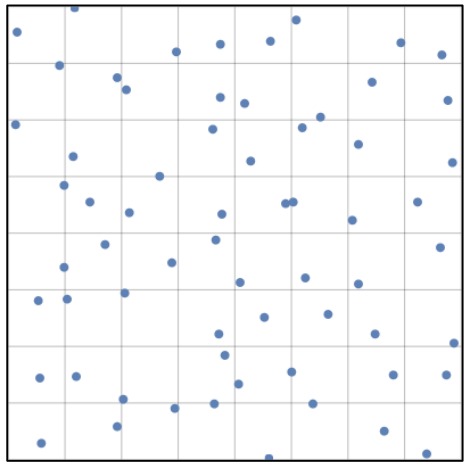
- Better strategy: stratified sampling (aka jittered sampling)
- If we divide the pixel area into a grid of $\sqrt{N} \times \sqrt{N}$ cells with one sample placed randomly within each cell that stratification gives us a variance of the mean of $O(1/N^3)$



(a) Random pattern



(b) Uniform stratified pattern

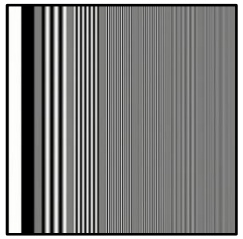


(c) Stratified jittered pattern

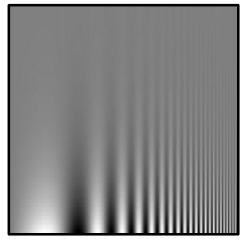
Source: Pharr M., Jakob W., Humphreys G. Physically Based Rendering: From Theory To Implementation. 2020.

Stratified Jittered Sampling

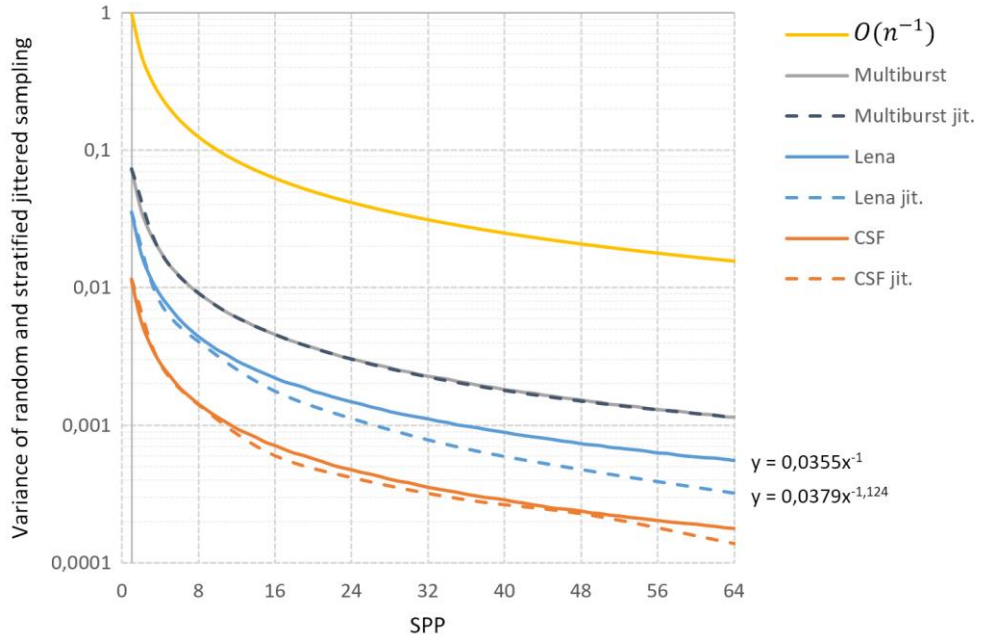
Three various contents of sampled pixel

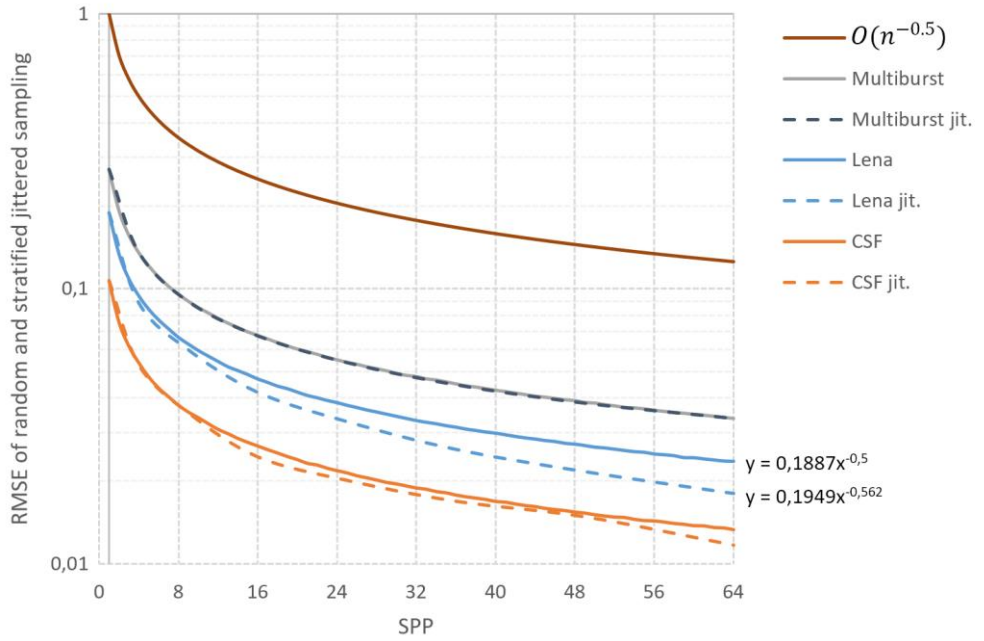

Multiburst


Lena


CSF

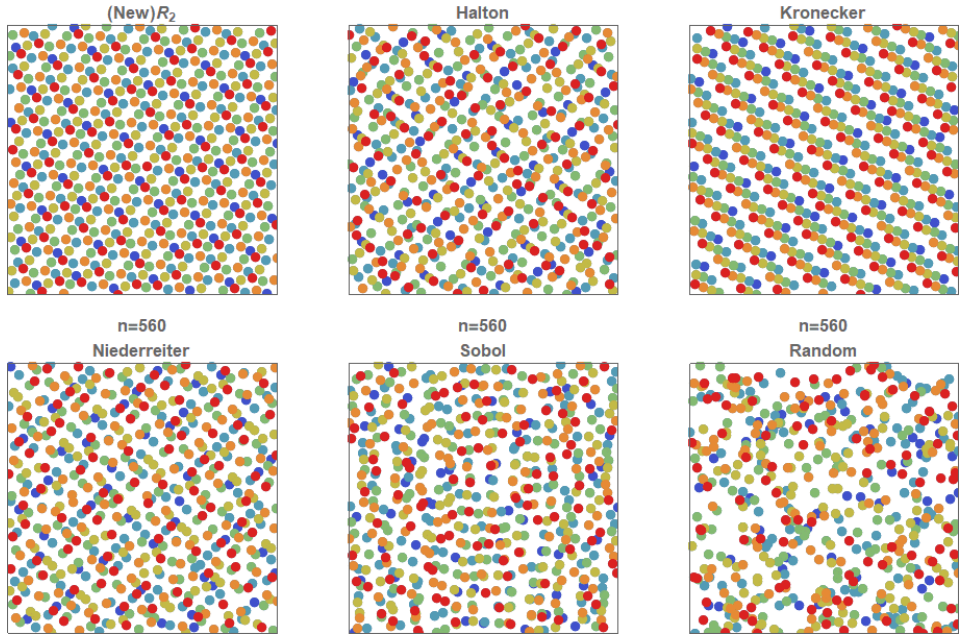
- Converges to the mean asymptotically faster than uniform random sampling
- The improvement depends on the nature of the image within pixel area
- In worst case, no better (but no worse) than uniform sampling





Super Sampling

- Even better strategy: low discrepancy quasirandom sequences (Halton, Niederreiter, Sobol, Hammersley) of points generated in a deterministic manner that reduces the likelihood of clustering (discrepancy) whilst still ensuring that the entire space is uniformly covered
- Methods require careful selection of basis parameters otherwise it can lead to degeneracy (e.g. top right)



Source: Roberts M. The Unreasonable Effectiveness of Quasirandom Sequences. 2020.

Pseudo/Quasi Random Numbers Generator

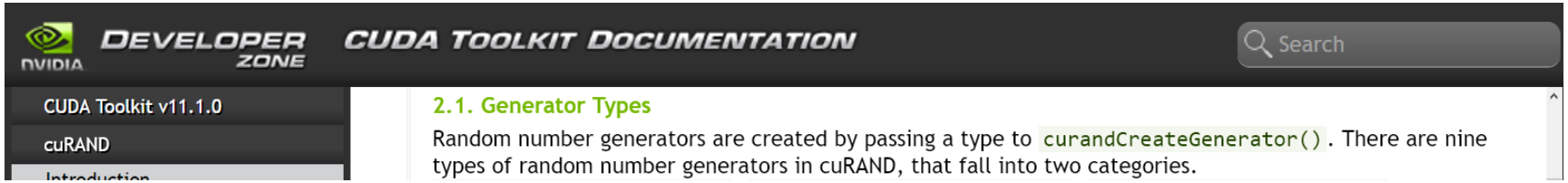
- Simple generator of a pseudo-random number uniformly distributed in the range $\langle a, b \rangle$

```
srand( 123 ); // random generator seed, call this only once!
float ksi = ( rand() / float( RAND_MAX ) ) * ( b – a ) + a;
```

- Don't use rand, use Mersenne Twister from <random> header instead

```
#include <random>
std::mt19937 generator( 123 );
std::uniform_real_distribution<float> unif_dist( a, b );
float ksi = unif_dist( generator );
```

- or try to use GPU (e.g. cuRAND, the CUDA random number generation library)

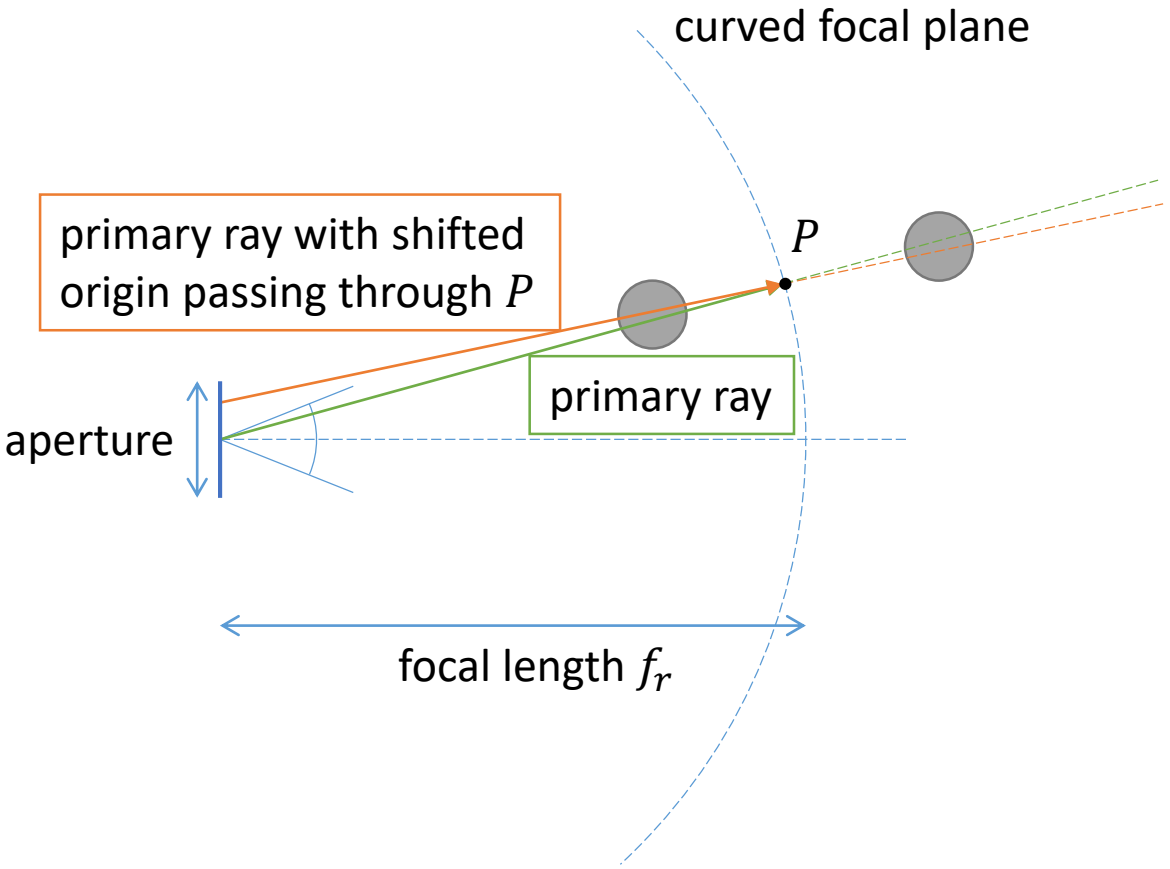


Defocus Blur and Depth of Field

Source: Marrs, A., Shirley, P., Wald, I. (ed.). *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*.

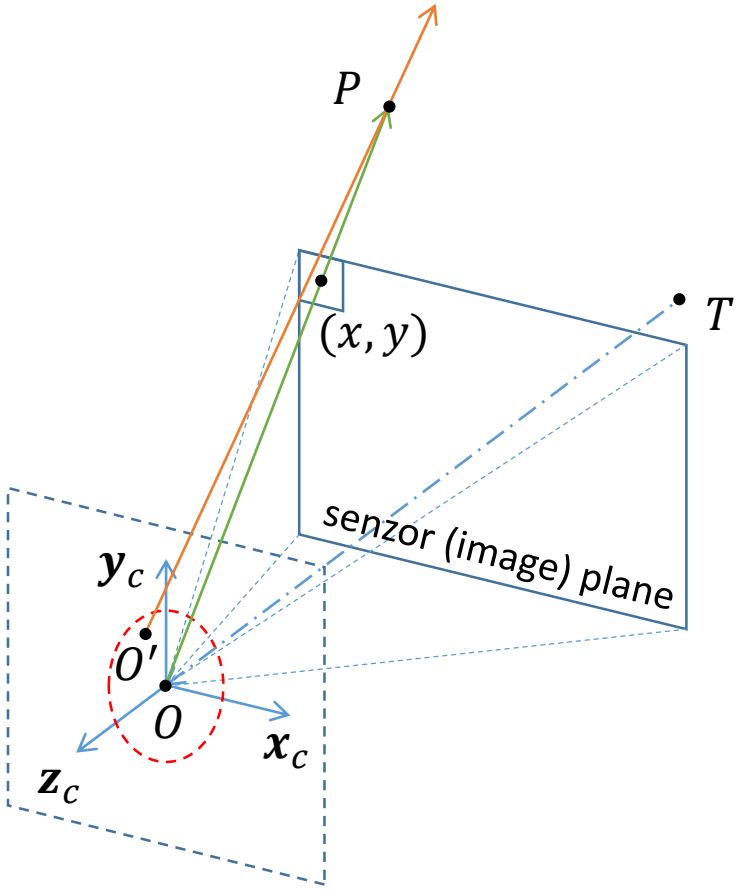
- **Defocus blur** is a blurring caused by the circle of confusion for any object not intersecting the focal plane.
- **Depth of field** is the distance between the two planes, one near and one far, wherein all objects appear sharp or in focus. All objects and features that do not perfectly intersect the plane of focus will experience defocus blur. A crude simplification is to say that a feature will only appear sharp if the circle of confusion is smaller than the pixel pitch of the sensor. Depth of field can sometimes colloquially be used to refer to defocus blur. This is wrong.
- ***Circle of confusion*** is an optical blur caused by the convolution of photon cones emanating from adjacent scene features. The radii of these photon cones are directly proportional to the aperture/iris.

Depth of Field



1. Define the radius f_r of focal sphere (focal plane is curved due to the Petzval field curvature)
2. Get the WS coordinates of point P which lies on the primary ray at the distance f_r
3. In random manner, shift the origin of the primary ray using the aperture size
4. Set the direction vector of the primary so that it goes through the focal point P.

Depth of Field



O (origin, view from) is the point from which we look

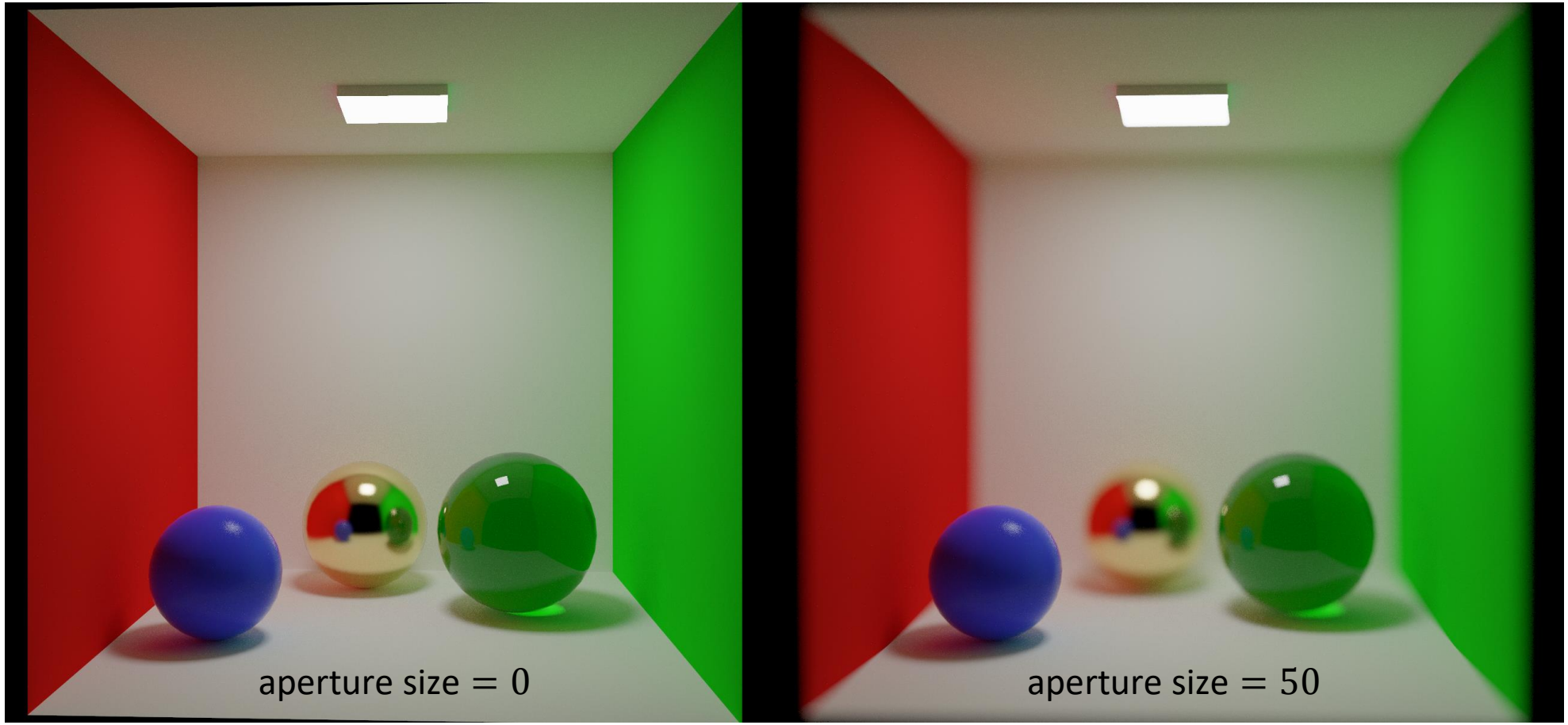
T (target, view at) is the point we are looking at

O' is a random point lying in the sensor plane (given by the axes $\boldsymbol{x}_c$ and $\boldsymbol{y}_c$) at a distance less than aperture size (radius of the red circle)

The green line represents the primary ray passing through the (x, y)-th pixel which is sampled randomly according to the chosen super sampling method with the point P lying on this ray at a distance f_r (distance/depth of maximum sharpness)

The orange line represents the slightly shifted primary ray which has a displaced origin O' and passes through the point P as well

Depth of Field



Depth of Field