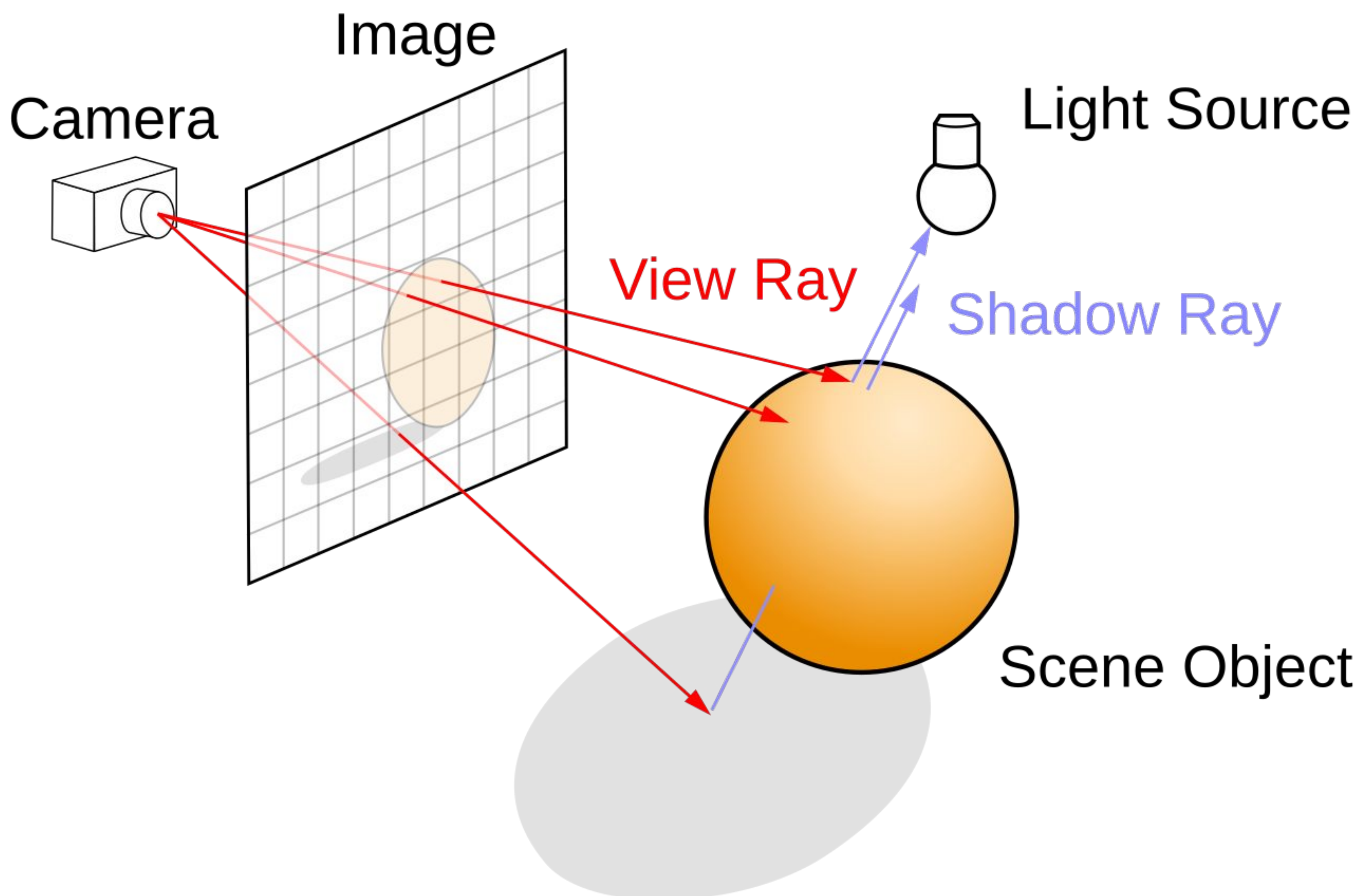
Background - Ray Tracing



The ray tracing algorithm is a way to render a 3D scene to a 2D image by simulating the mathematical properties of light rays.

*Image source: Nvidia Developer

Basic Algorithm of Ray Tracing

Parallelizable

```
for pixel in image:
    for sample in range(num_samples_per_pixel):
        ray = MonteCarloSampling(pixel_coordinate)
        pixel_color += ray_color(max_depth, ray, world, lights)
    return pixel_color / num_samples_per_pixel
```

Rendering Algorithm

```
def ray_color(depth, ray, world, lights):
    # If the recursion depth is zero or less, return black
    if depth <=0:
        return Color(0, 0, 0)

    # Check if the ray hits anything in the world
    if ray hits anything in world:
        material = HitRecord()
    else:
        return background_color # Return the background color if no hit

    # Compute the emitted color from the material
    emitted_color = material.emit() # Only light source emit color

    # Check if the material scatters the ray
    if material.scatter():
        attenuation, scattered_ray = ScatterRecord()
    else:
        return emitted_color  # Return emitted color if no scattering

    return emitted_color + attenuation * \
        ray_color(depth - 1, scattered_ray, world, lights)
```

Recursive Ray Color on CPU

```
def ray_color(depth, ray, world, lights):
    final_color = color(0, 0, 0)
    cur_attenuation = color(1, 1, 1)
    cur_ray = ray

    for i in range(depth):
        # Check if the ray hits anything in the world
        if cur_ray hits anything in world:
            material = HitRecord()
        else:
            # Add background color and return if no hit
            final_color += cur_attenutaion * background_color
            return final_color

        # Compute the emitted color from the material
        emitted_color = material.emit() # Only light source emit color

        # Check if the material scatters the ray
        if rec.material.scatter():
            attenuation, scattering_ray = ScatterRecord()
        else:
            # Add emitted color and return if no scattering
            final_color += cur_attenutaion * emitted_color
            return final_color

        # Add emited color to final color
        final_color += cur_attenuation * emmited_color
        # Update current attenuation and current ray
        cur_attenuation *= attenuation
        cur_ray = scattered_ray

    return final_color
```

Iterative Ray Color on GPU

*The Ray Color Algorithm is simplified

Parallel Ray Tracing - From Architecture Perspective

Motivation: Each ray operates independently for each pixel, enabling inherent parallelism across a large number of pixels.

Multi-core CPU (OpenMP)

- Utilizes Multiple Cores for task-level parallelism by dividing the workload across available CPU cores.
- Advantages:
  - **Ease of Programming:** OpenMP pragmas simplify the implementation of parallel loops for pixel-based tasks.
- Challenges:
  - **Low Parallelism:** Parallelism is primarily confined to task-level (coarse-grained across cores) or control flow-level (exploited by Out-of-Order execution), with minimal ability to leverage fine-grained data-level parallelism compared to GPUs.

GPU (CUDA)

- Uses SIMT to process thousands of pixels concurrently.
- Advantages:
  - **Massive Thread Parallelism:** Executes thousands of CUDA threads simultaneously, achieving high throughput.
- Challenges:
  - **Branch Divergence:** Threads in a warp may follow different execution paths (e.g., varying ray reflections or BVH traversal), reducing performance.
  - **Memory Divergence:** Poor memory coalescing can degrade performance due to non-coalescing data access patterns.

BVH Acceleration - Binary Search of Hittable Objects

```
def world_hit(ray):
    # Need to know the closest object that hits
    closest_hit = None
    for object in world:
        if ray hits object and object is closer than closest_hit:
            closet_hit = object
    if closest_hit is not None:
        # Record the detail of the final hit detail
        rec = closest_hit.record()
        return True
    return False
```

Naive Hit

```
def bvh_node_hit(ray):
    if ray hits my_bounding_box:
        hit_left = left_child.bvh_node_hit(ray)
        hit_right = right_child.bvh_node_hit(ray)
        return hit_left or hit_right
    else:
        return False
```

Recursive Hit on CPU

```
def bvh_node_hit(ray):
    stack.push(root)
    while stack not empty:
        current = stack.pop()
        if ray hits current.bounding_box:
            stack.push(left_child)
            stack.push(right_child)
        else:
            continue
```

Iterative Hit on GPU

Reduce Search Time of Hit from O(N) to O(logN) !

*The bvh hit function is simplified

BVH Construction

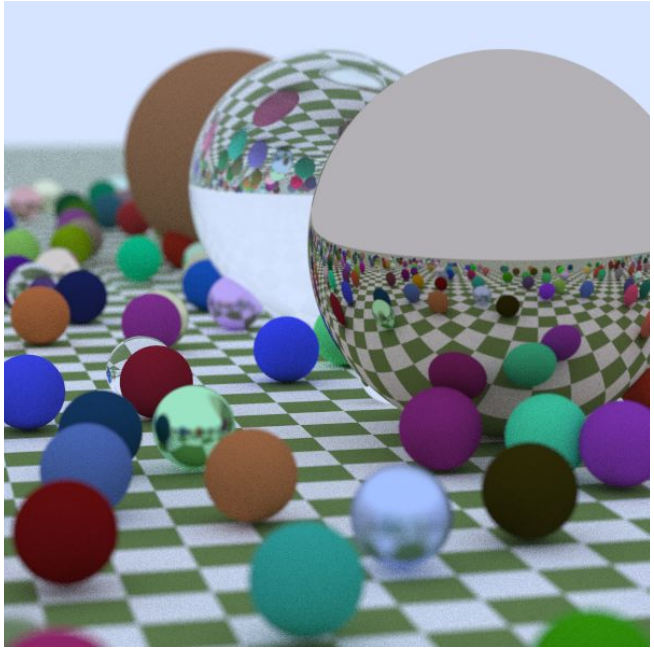
```python
def create_bvh_node(objects, start, end):
    # Create a bounding box that can hold all the objects
    bounding_box = CreateBoundingBox(objects[start:end])
    # Create child nodes
    object_span = end - start
    # This adds actual object as child
    if object_span == 1:
        left_child = right_child = objects[start]
    elif object_span == 2:
        left_child, right_child = objects[start], objects[end]
    # This adds new bvh node as child
    else:
        # Divide at the longest axis
        sort_objects(longest_axis)
        mid = start + object_span / 2
        left_child = create_bvh_node(objects, start, mid)
        right_child = create_bvh_node(objects, mid, end)
    return bvh_node(left_child, right_child)
```

Recursive Construction on CPU

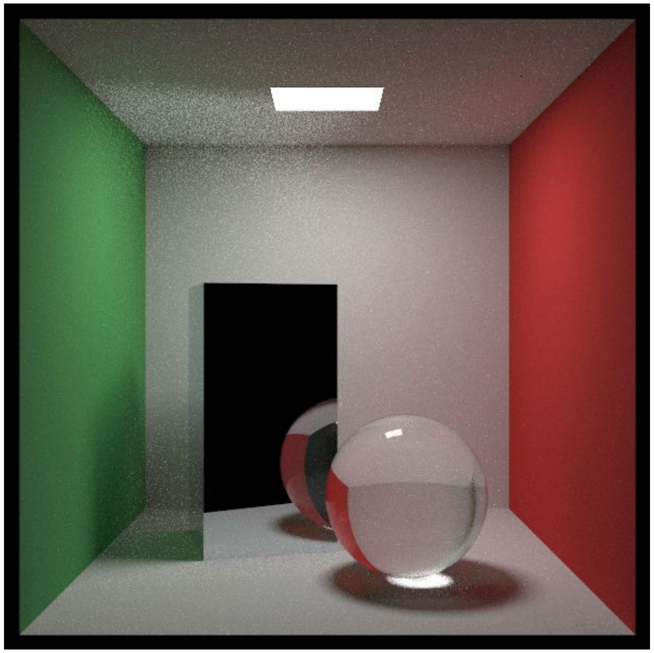
```python
def create_bvh_node(objects, start, end):
    root = bvh_node(None, None)
    stack.push((root, start, end))
    while stack not empty:
        current, cur_start, cur_end = stack.pop()
        # Create a bounding box that can hold all the objects
        bounding_box = CreateBoundingBox(objects[cur_start:cur_end])
        # Create child nodes
        # This adds actual object as child
        if object_span == 1:
            left_child = right_child = objects[start]
        elif object_span == 2:
            left_child, right_child = objects[start], objects[end]
        # This adds new bvh node as child
        else:
            # Divide at the longest axis
            sort_objects(longest_axis)
            mid = start + object_span / 2
            current.left_child = bvh_node(None, None)
            stack.push((current.left_child, start, mid))
            current.right_child = bvh_node(None, None)
            stack.push((current.right_child, mid, end))
    return root
```

Iterative Construction on GPU

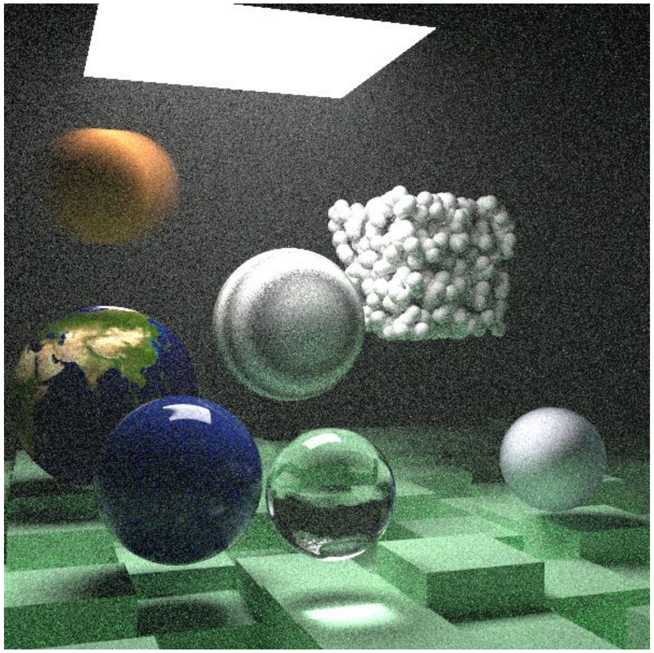
Four Scenes of Rendered Image



First Scene (488 Objects)



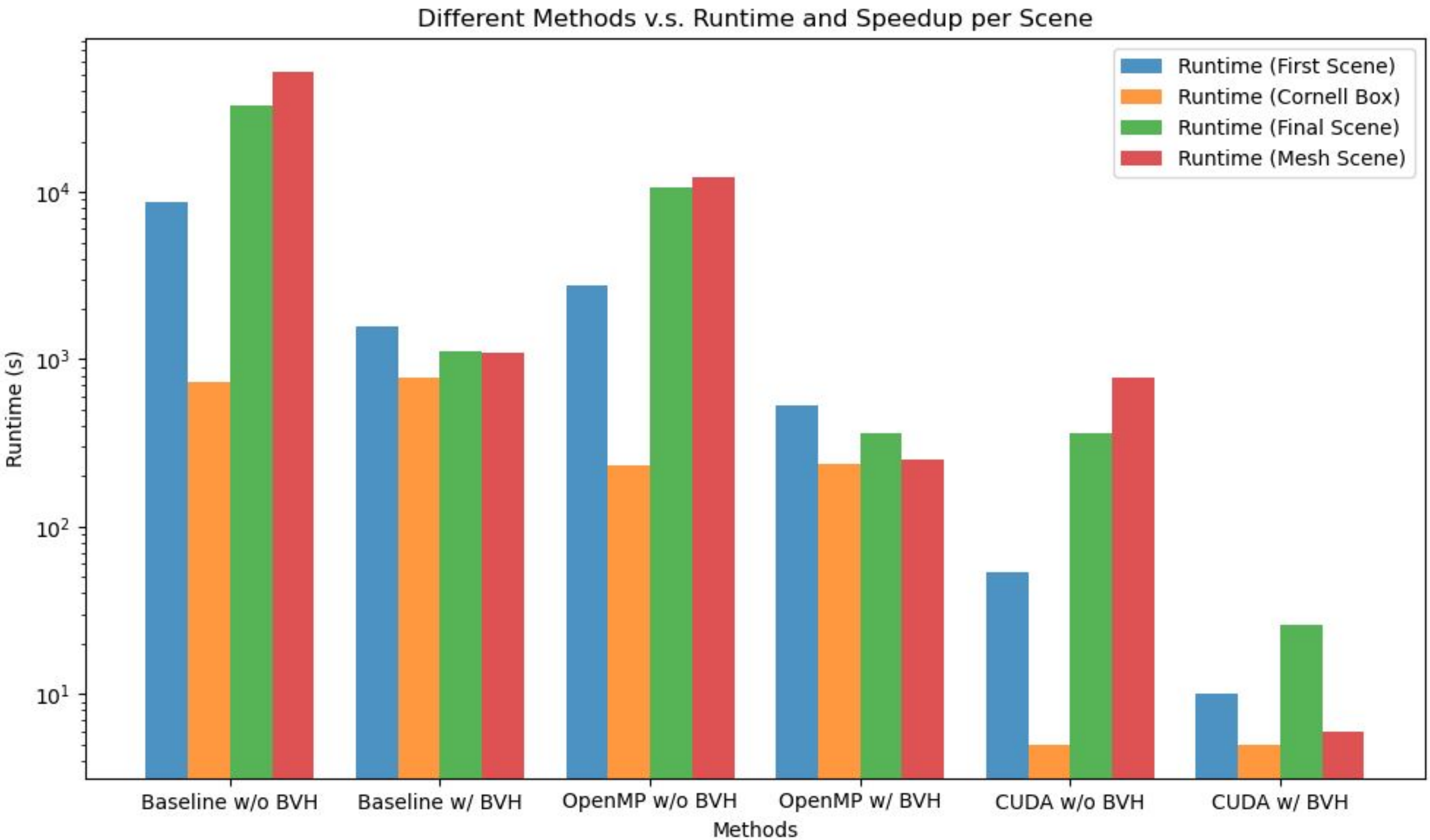
Cornell Box (13 Objects)



Final Scene (3409 Objects)

Mesh Scene (4974 Objects)

Runtime of Different Methods on Each Scene



In Mesh Scene, we can achieve 8612x speedup

More Results