Assignment 3 Overview

Computer Graphics
CMU 15-462/662

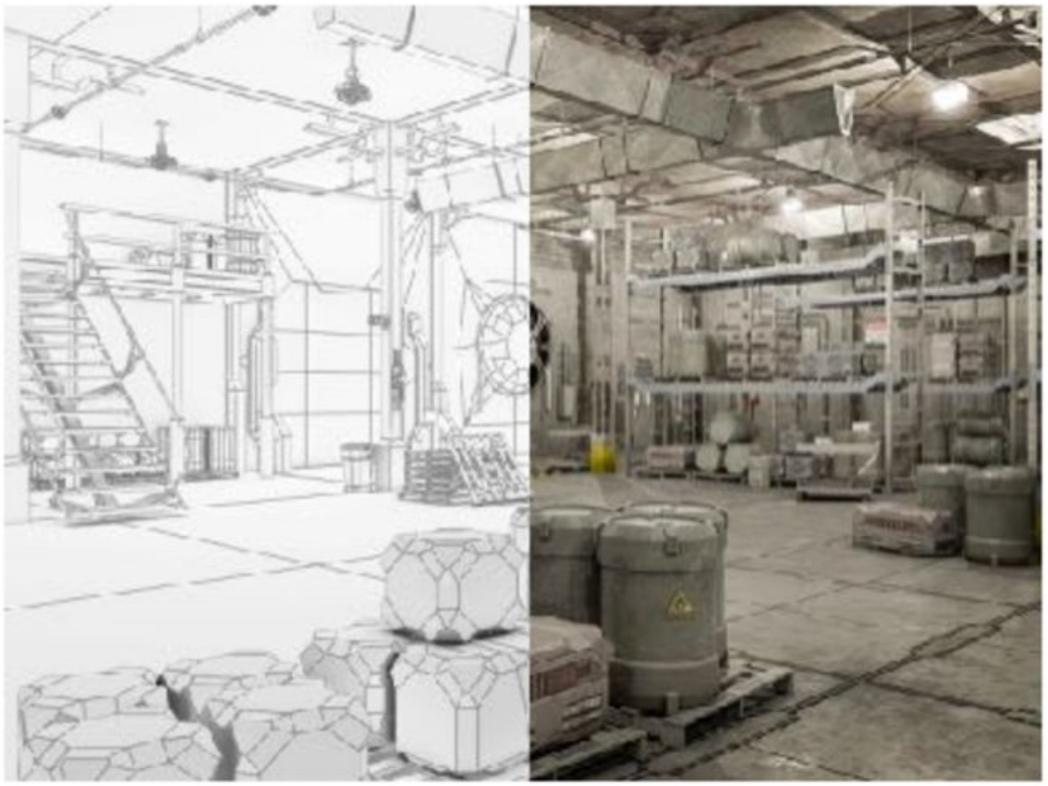
Logistics

- Midterms have been graded, collect them in class today
- Each *page* (not problem) has been graded by a different TA. Contact the individual TA for regrade requests.
  - Page 1: Yuqiao
  - Page 2: Connor
  - Page 3: Zach
  - Page 4: Adrian
- Mid-semester letter grades were calculated based on Assignment 0.0, Assignment 0.5, DrawSVG, and the midterm exam.

Assignment 3: Pathtracer

- Extension of the work you did in MeshEdit
  - Now that we can create meshes with Scotty3D, it's time to build a renderer that computes a realistic rendering of the scene
- Warning: Pathtracer will be difficult for different reasons than MeshEdit was difficult!
  - In MeshEdit we aimed to maintain the invariants of a complex data structure
    - Errors are more "obvious" and result in crashes/hangs
  - In PathTracer, we aim to maintain physical accuracy, but we aren't changing the scene at all
    - Errors are related to math or theory and the symptoms are usually visual and may not be obvious

Rasterization vs Pathtracing



Rasterization

Transform scene geometry via matrix operations to screen space, then use triangle fill algorithm.

Optimized for performance

DrawSVG (A1)

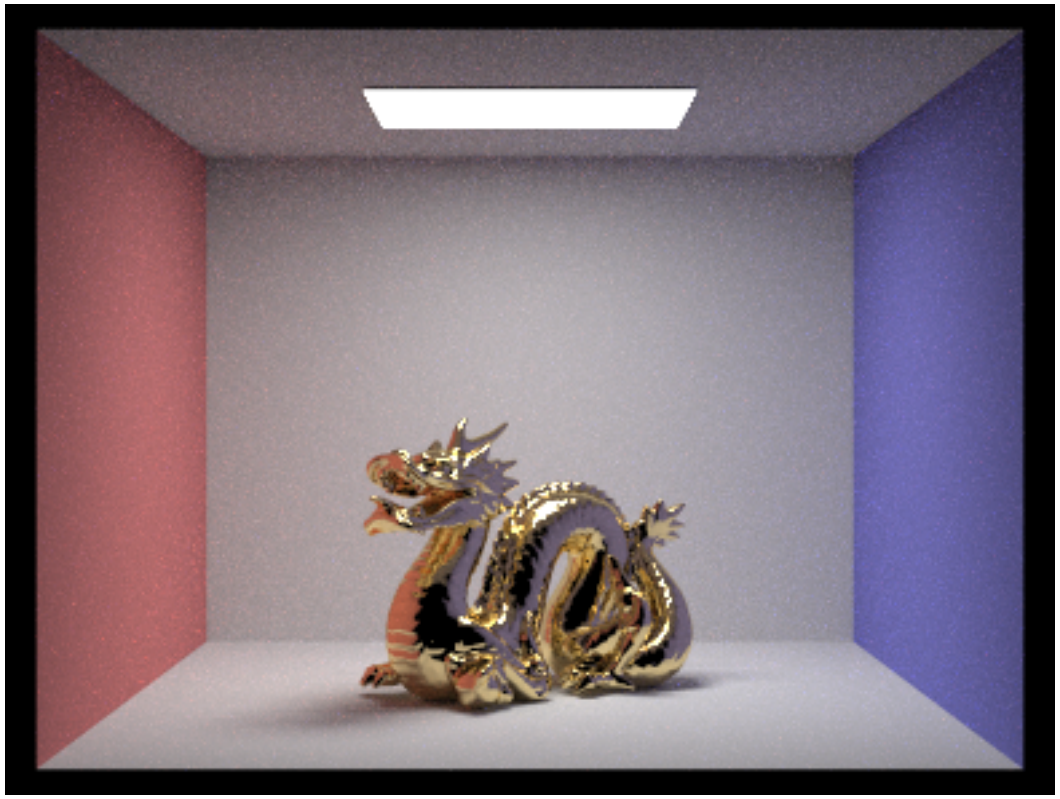
Pathtracing

Bounce simulated rays of light throughout your scene randomly for each pixel, and illuminate if it eventually intersects a light.

Optimized for realism

Pathtracer (A3)

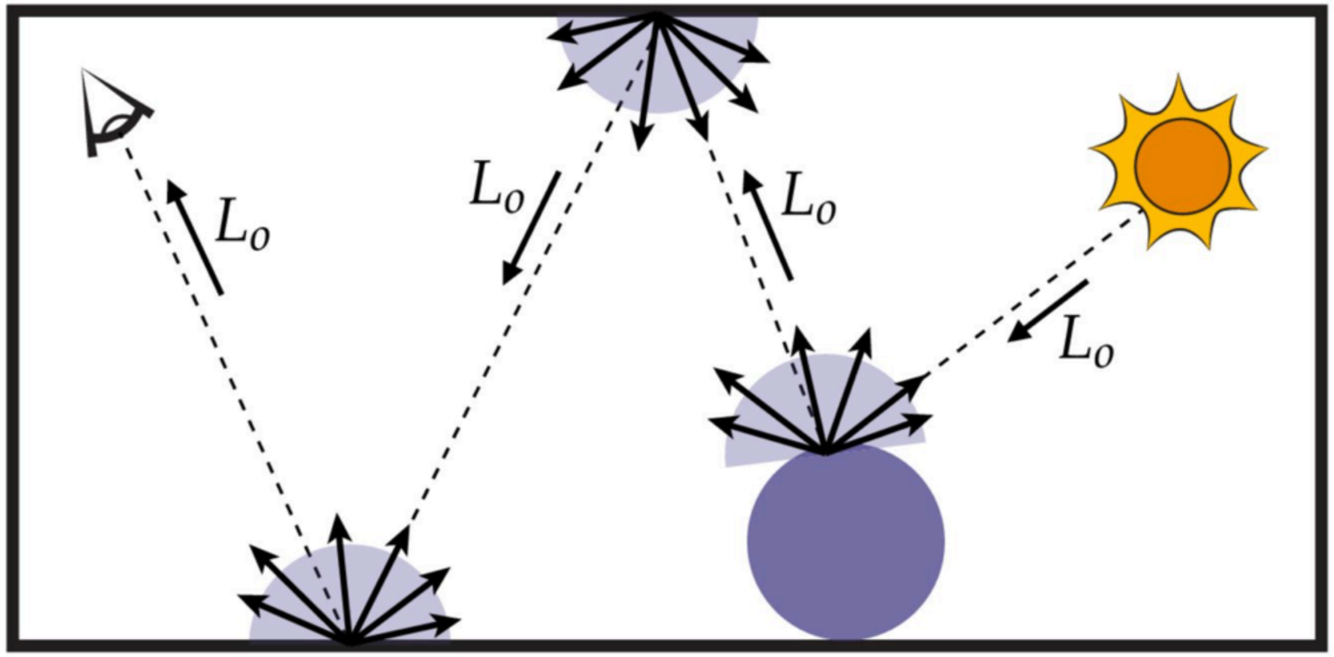
Example Scotty3d Output



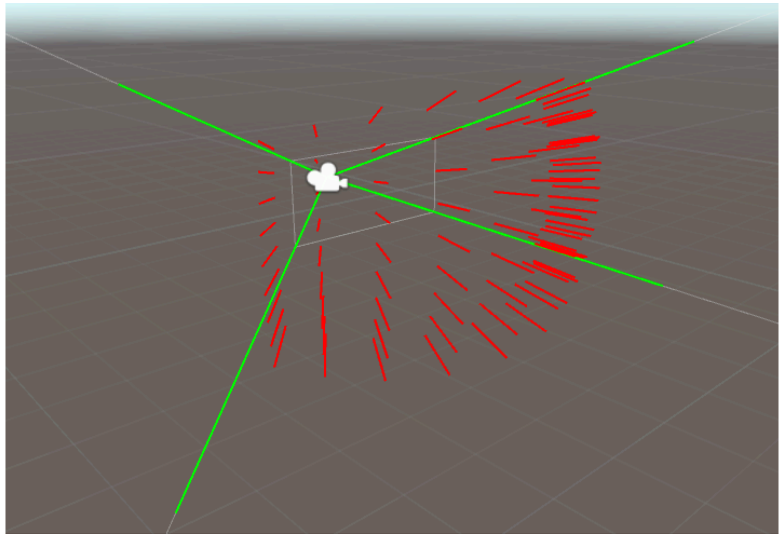
The Big Picture

End Goal

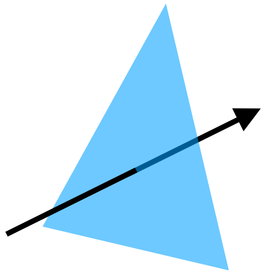
- You are tasked with building a *pathtracer*, which simulates rays of light bouncing around your scene and eventually "into the camera"
  - (Small detail: we will actually "start" our rays at the camera origin and bounce it around the scene until we hit a light)
  - Over the next few weeks we will dive into the physics of Color, Radiometry, the "Rendering Equation," and more details that are important in designing a pathtracer
  - Up to this point you are ready to complete the assignment up to + including Task 4 (shadow rays).



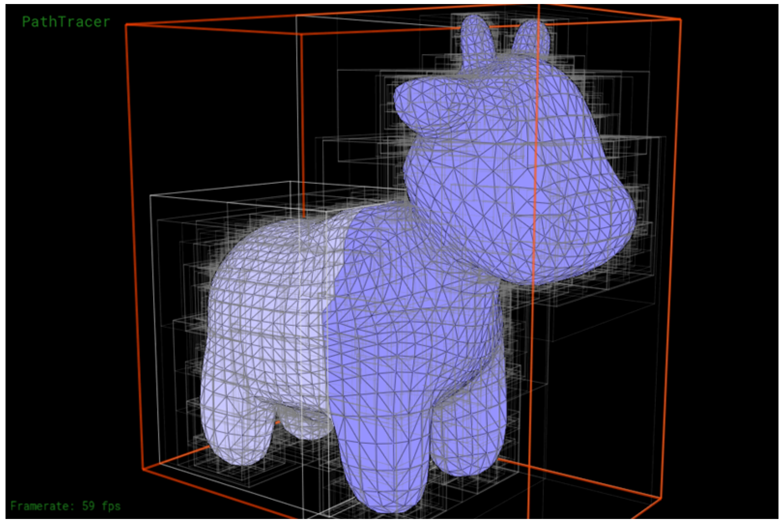
Overview of Tasks



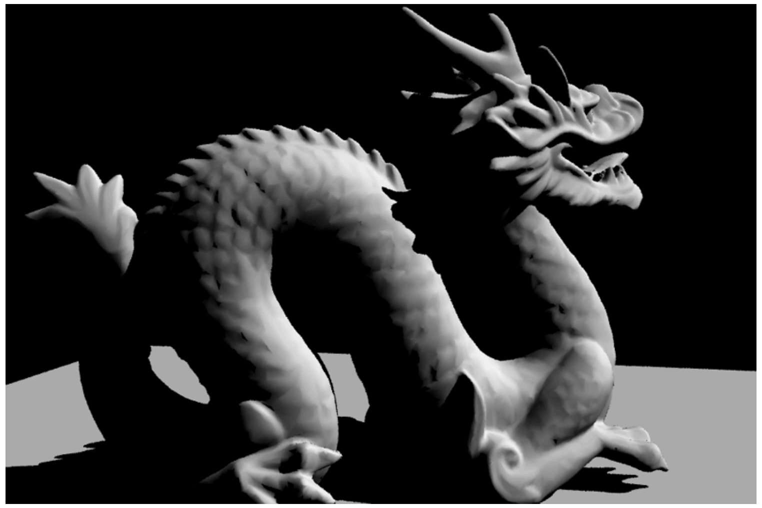
- Task 1: Generate the initial rays to send from the camera
- Task 2: Compute ray-primitive intersection
  - You need to support triangles and spheres



- Task 3: Accelerate ray-scene intersection queries using a Bounding Volume Hierarchy (BVH)

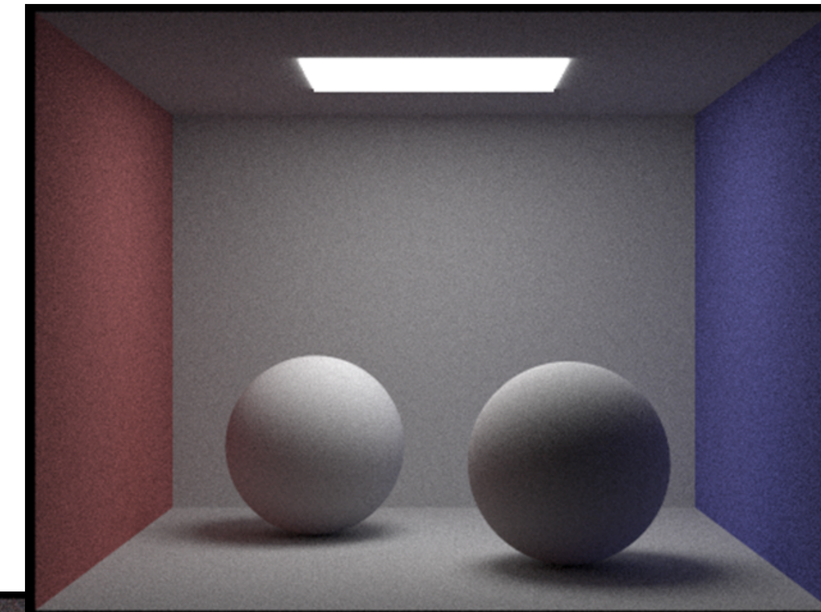


- Task 4: Implement direct lighting with shadows

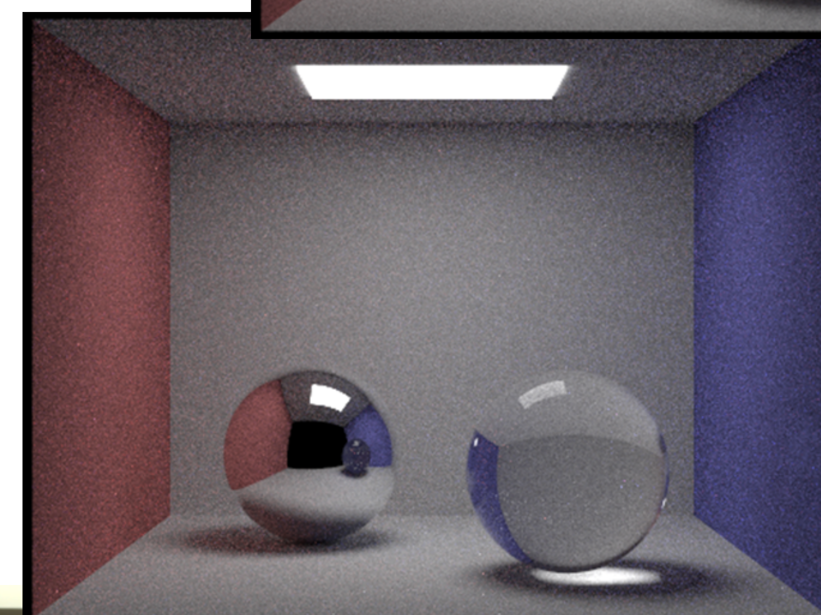


Overview of Tasks

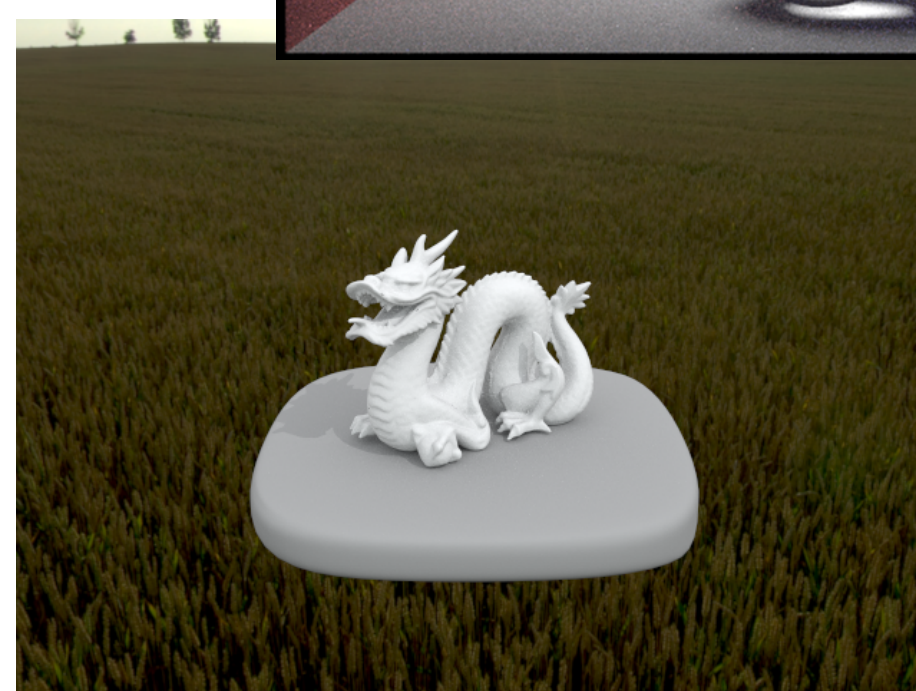
- Task 5: Support indirect illumination via path tracing



- Task 6: Support non-diffuse materials (mirror, glass)



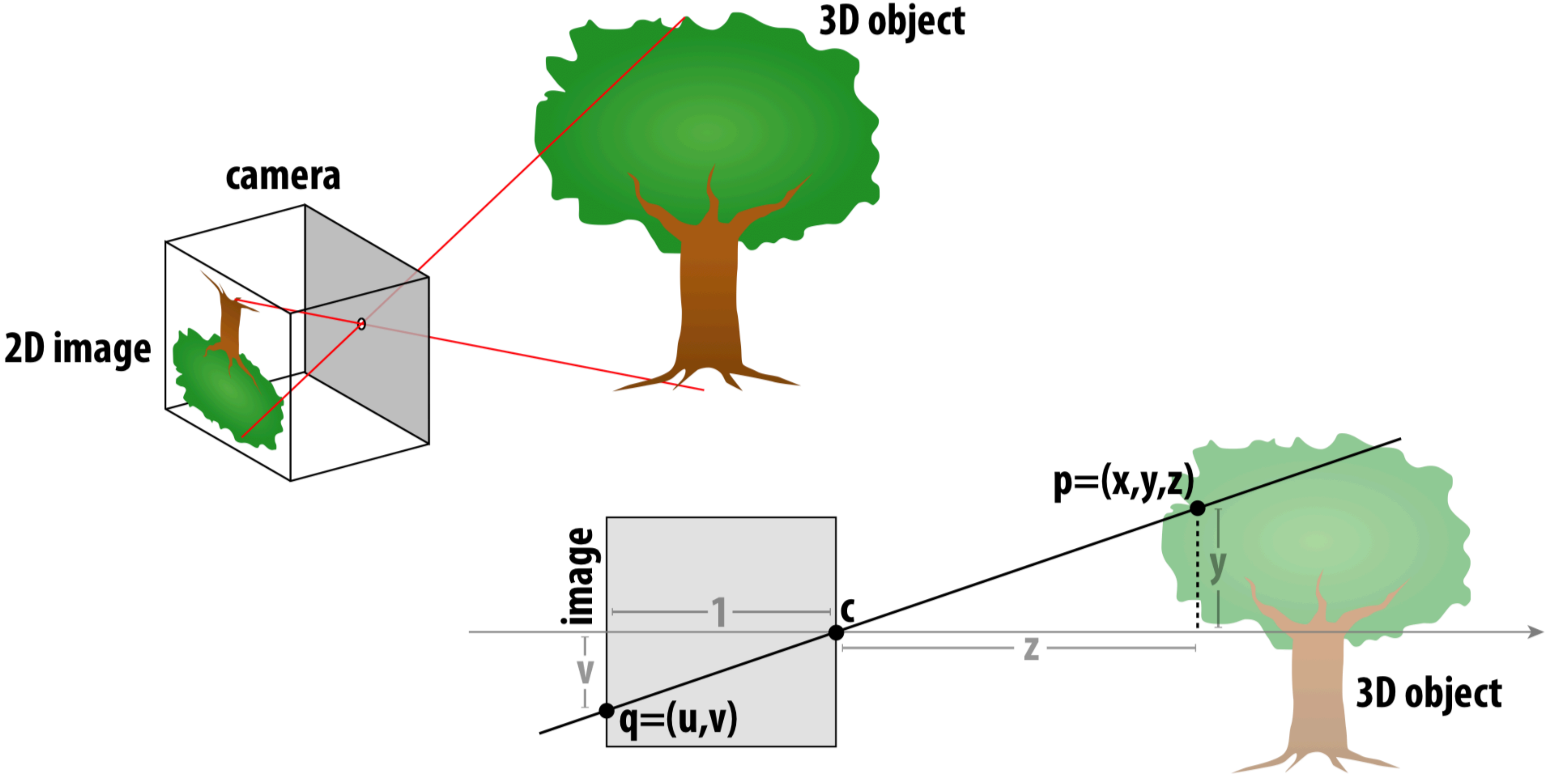
- Task 7: Support environment lighting via a texture



Camera Rays

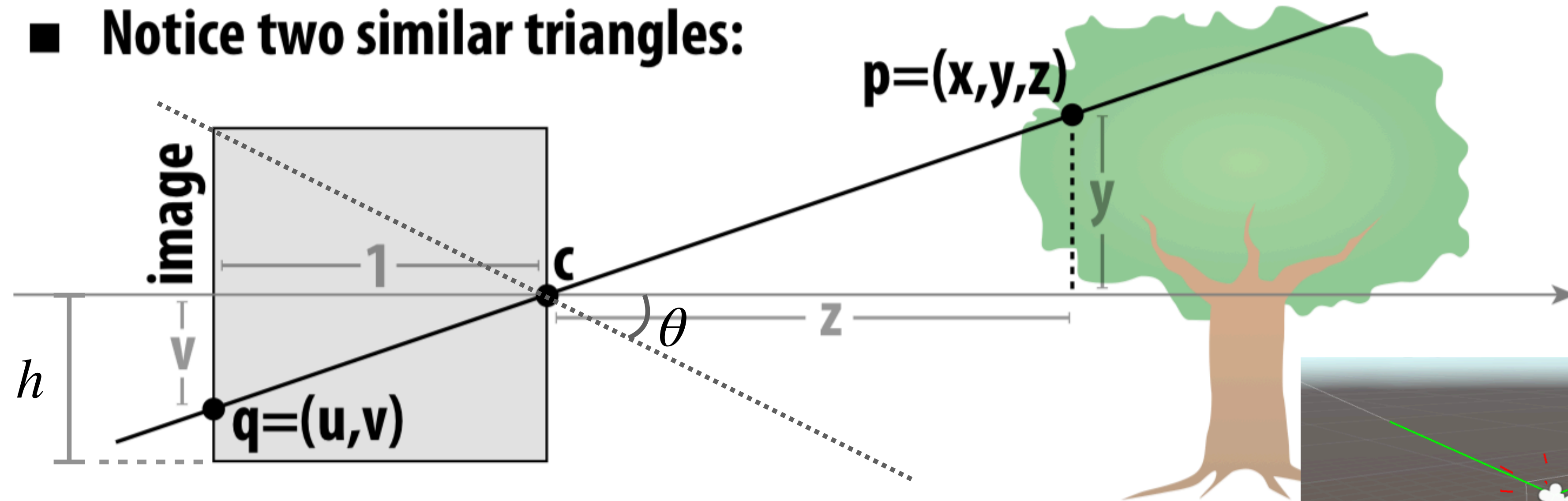
Recall Lecture 1: Perspective Projection

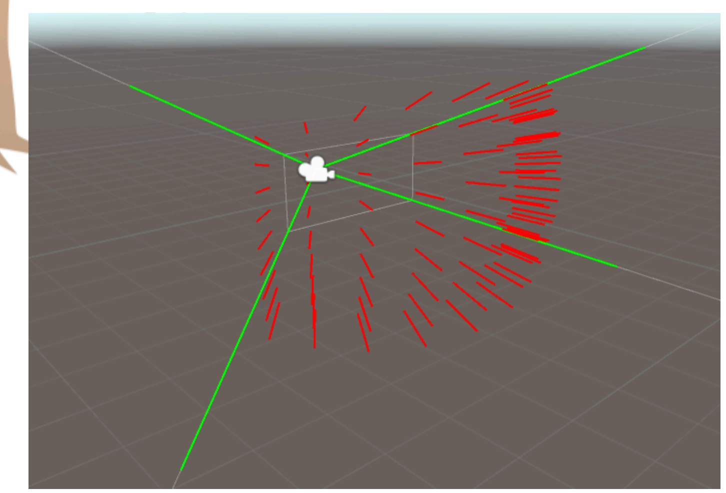
- In lecture 1 we considered the *pinhole* model for cameras:



Recall Lecture 1: Perspective Projection

- Notice two similar triangles:

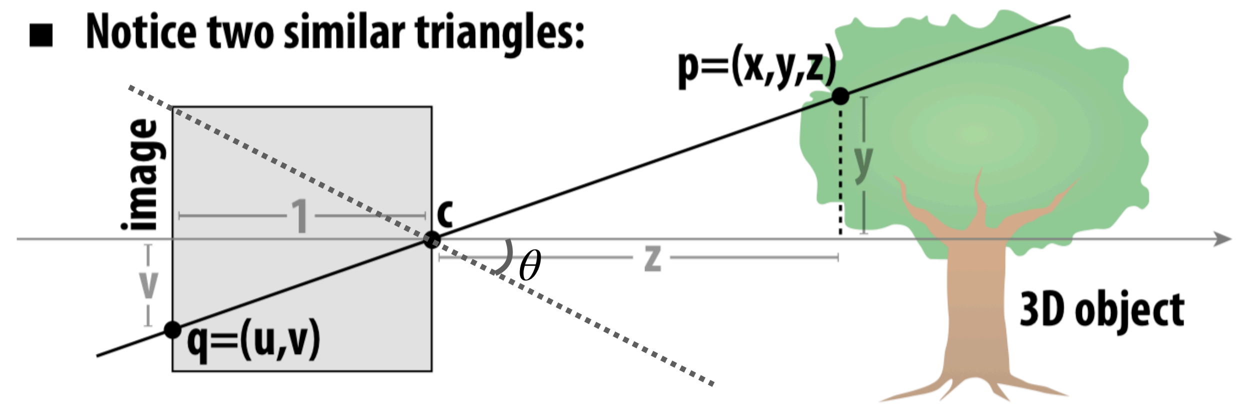




- ***Question***: What is h, given θ?
  - 2θ is the <u>vertical field of view</u>
  - v is the projection of p_y on the image plane, with extents $[-h, h]$
- Goal of part 1: generate the ray $\overrightarrow{cp}$ **in world space** given the camera position, orientation, and (u, v) coordinate
  - uvs are given in $[0,1]$ range, not $[0, h]$

Implementation

- Where is each variable in the figure, in camera.h?
  - Do we need anything more than the excerpt on the bottom right?
- Suggestion: Calculate the camera ray in camera space first
  - Take advantage of similar triangles — $y/v = ??$
  - How do we convert a camera space vector to world space?
  - Camera space: camera forward vector is $(0,0,1)$ and camera origin is at $(0,0,0)$ - much easier to reason about camera rays!




```
// camera.h
class Camera {
 public:

  // ...
  double v_fov() const { return vFov; }          // !!
  double aspect_ratio() const { return ar; }     // !!
  // ...

  // worldspace -> cameraspace transformation matrix
  Matrix4x4 getTransformation();                 // !!

  // Task 1
  Ray generate_ray(double x, double y) const;

  // ...
};
```

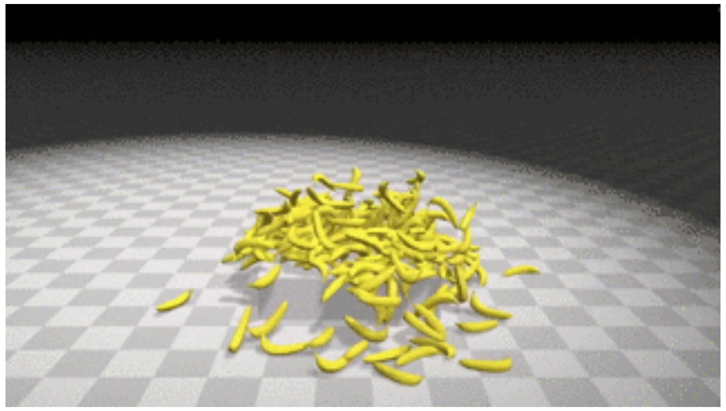
Bounding Volume Hierarchies (BVHs)

Recall: Spatial Data Structures

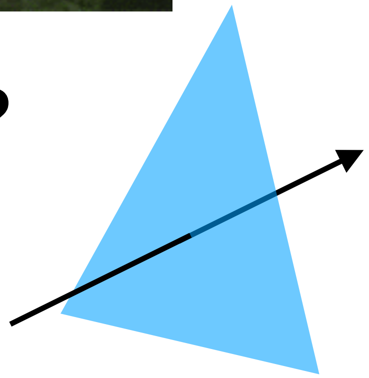
- Problem: *I want to efficiently perform a **query** on **primitives** that are ordered/arranged spatially. This query is only going to be relevant locally to some volume of space, and we would like to bail out of the computation early for "far-away" primitives that are outside that volume.*
  - Examples: Collision detection, frustum/occlusion culling

Left:
https://youtu.be/-S8wq0dz1H4

Right:
https://youtu.be/VqH8kcmD-HI



  - How do we cheaply figure out when to bail out?



- What's the query in raytracing? The primitive?
  - In Scotty3D: Ray-Triangle / Ray-Sphere intersection

Recall: Bounding Volume Hierarchy

- Divide all of your primitives into a hierarchy: a binary tree
  - The leaves are individual primitives
  - The nodes are *bounding volumes*

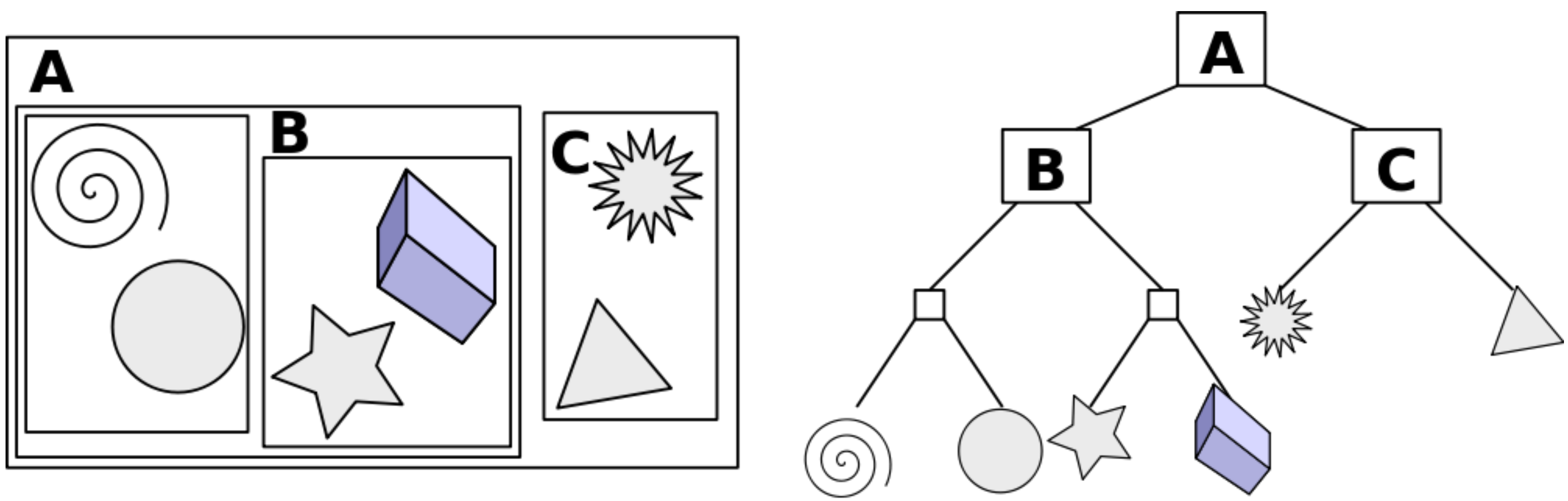
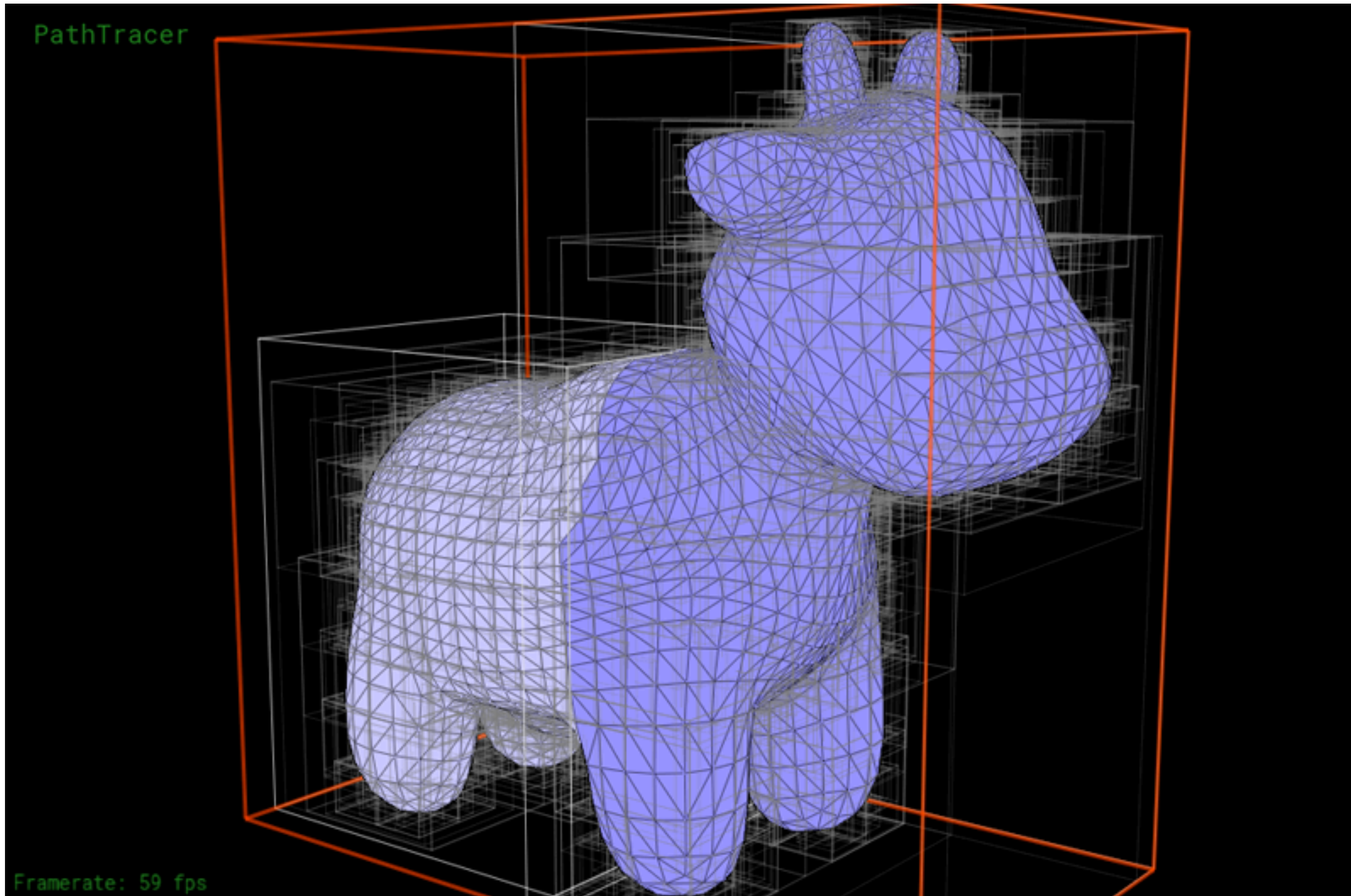


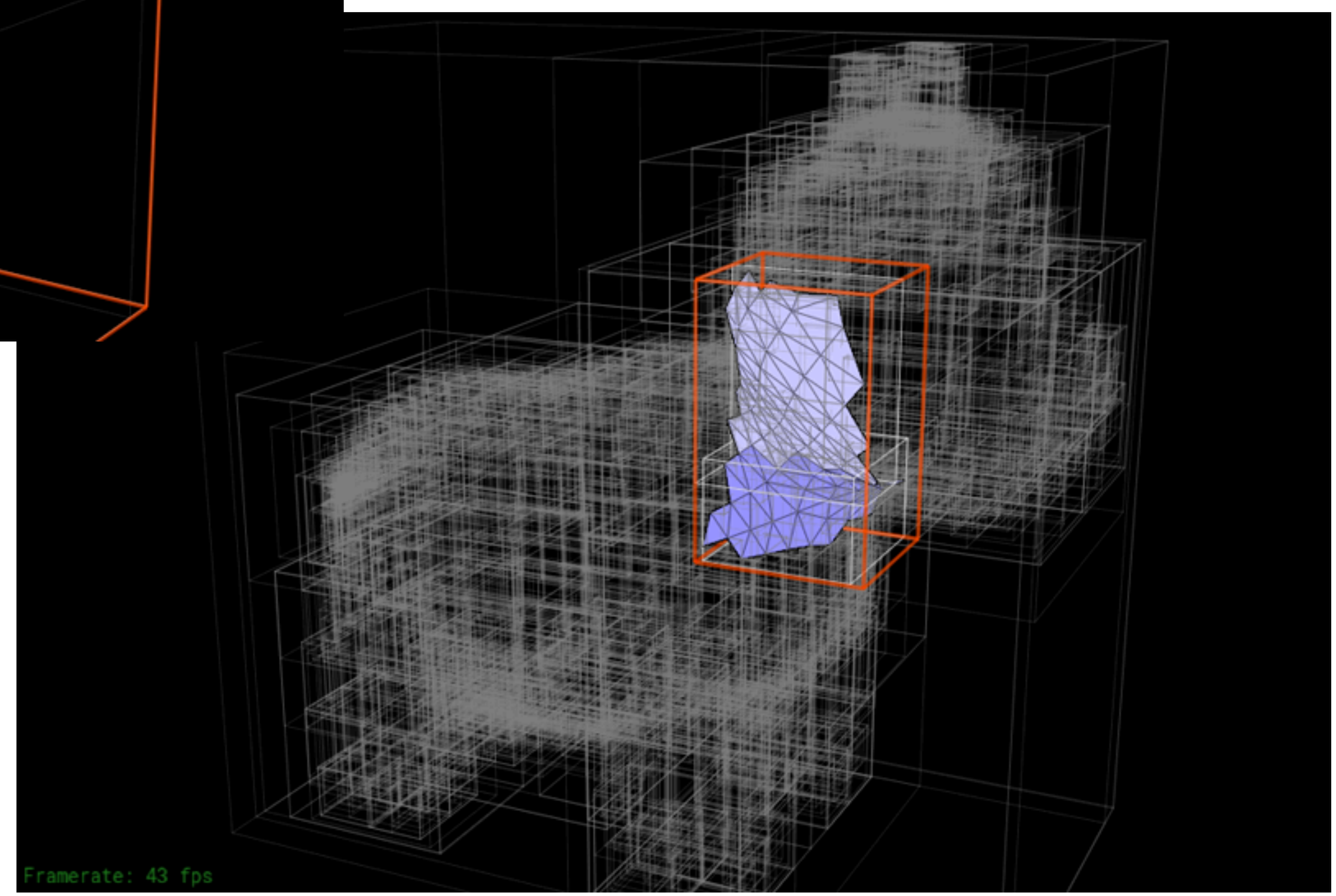
Image Source: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy

BVH in Scotty3D



Red box: Currently selected node
Dark blue triangles: “right” subtree
Light blue triangles: “left” subtree

Note: “right” / “left” does not have to do with the spatial positioning, only the topology of the BVH graph!

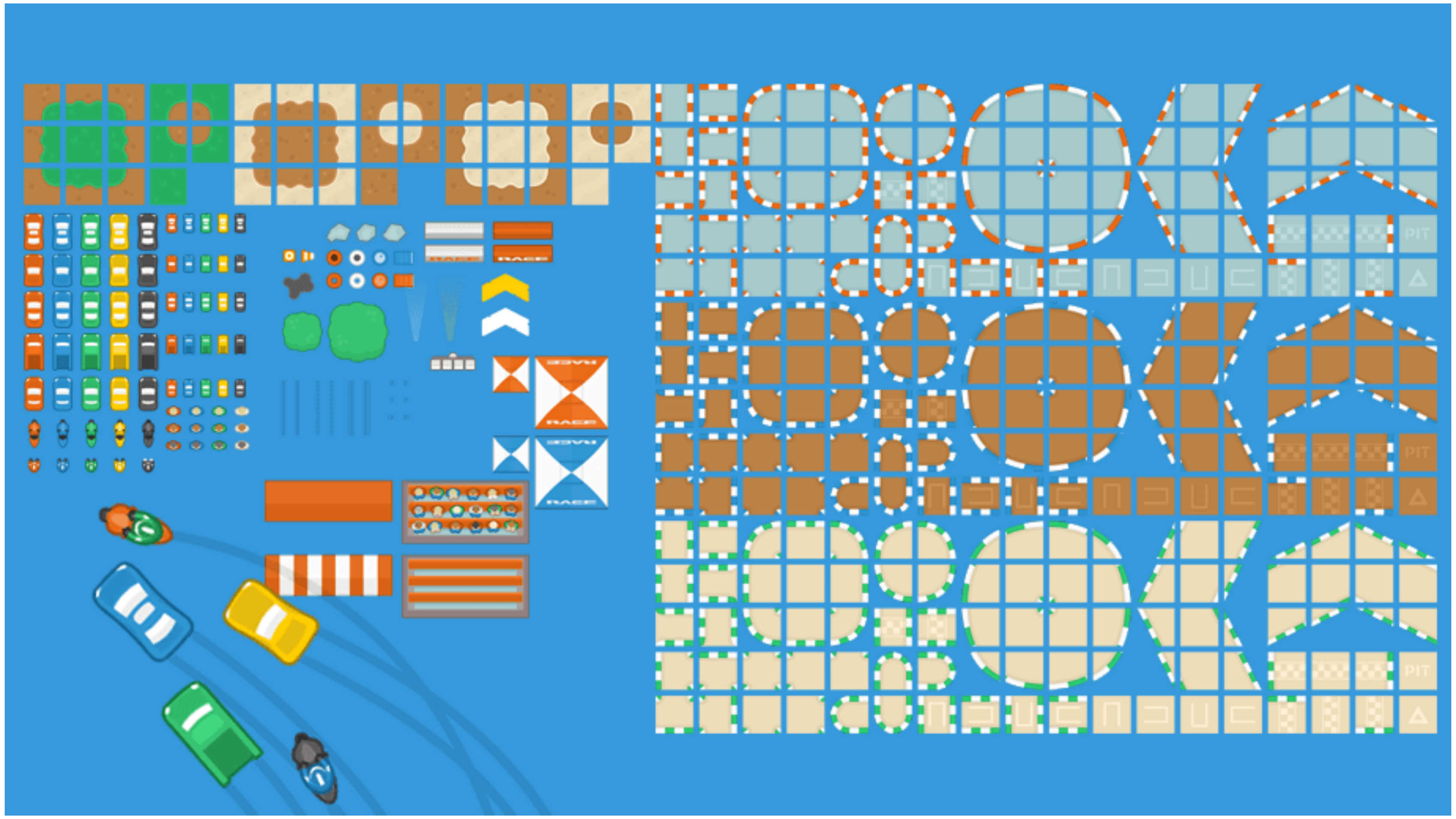


Once you’ve implemented BVH, you can look at this visualization via the V key after rendering

Press the < and > keys to descend the tree and ? to move to the parent

Exercise: Bounding Circle Hierarchy

- Suppose you're a graphics engineer working on a 2D video game in which you need to draw many thousands of vector graphics (like DrawSVG) on screen at a time. To speed up rendering, you propose using vector objects (polygons, curves, etc) as the primitives and circles as the volumes



Vector Object Assets (the primitives)

Source: https://kenney.nl/assets/racing-pack

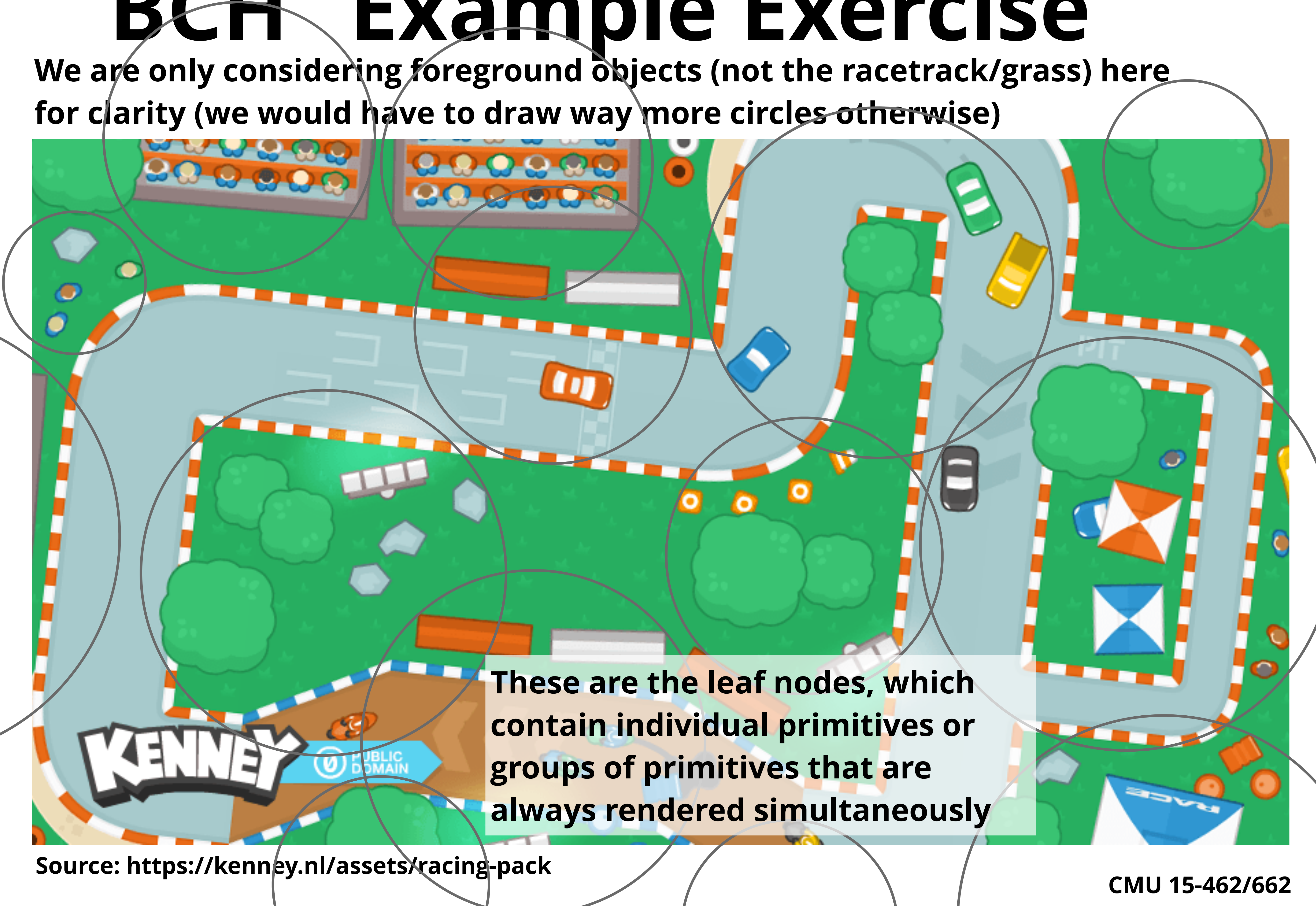
"BCH" Example Exercise

We are only considering foreground objects (not the racetrack/grass) here for clarity (we would have to draw way more circles otherwise)

Source: https://kenney.nl/assets/racing-pack

“BCH” Example Exercise

We are only considering foreground objects (not the racetrack/grass) here for clarity (we would have to draw way more circles otherwise)

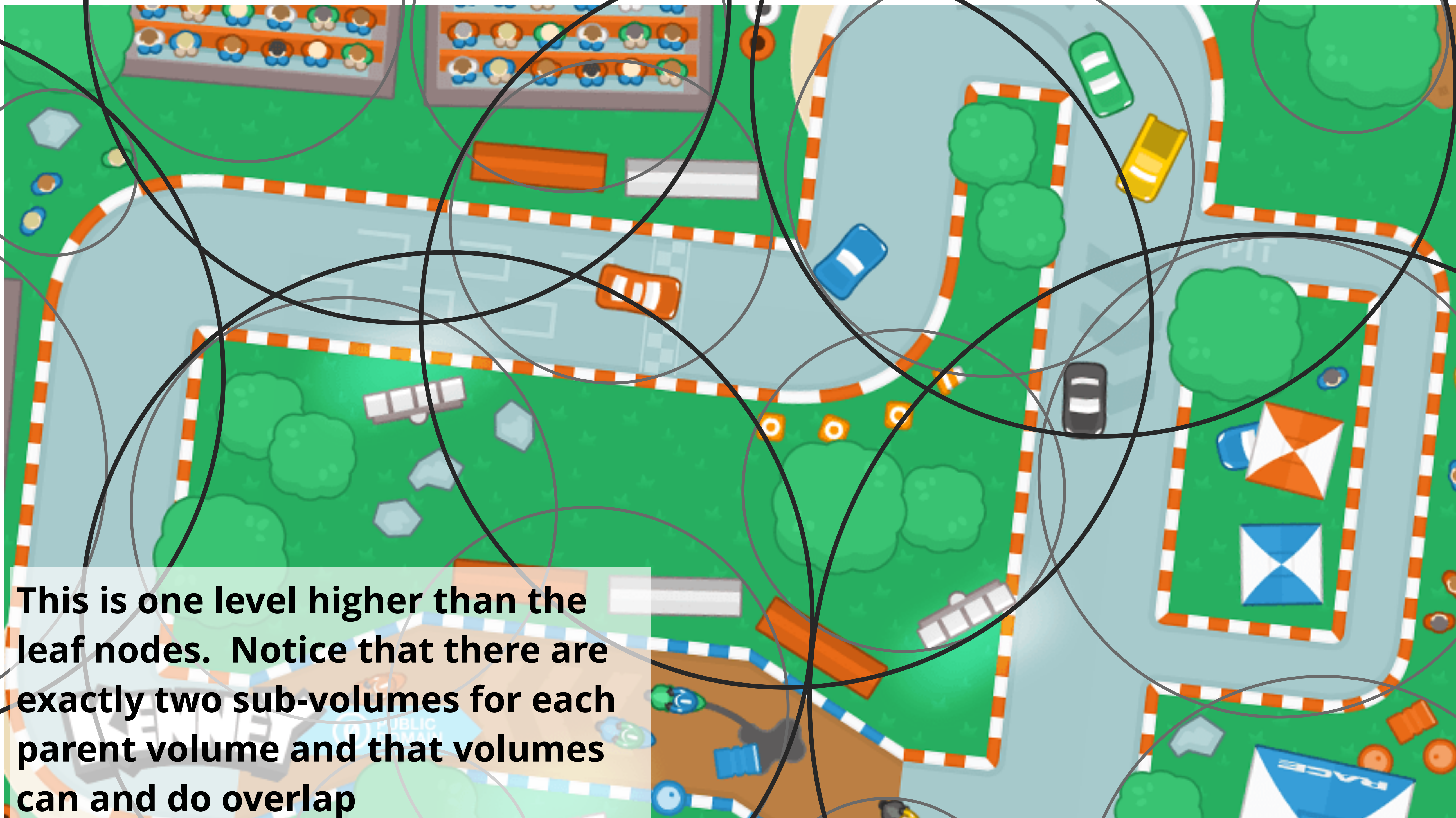


These are the leaf nodes, which contain individual primitives or groups of primitives that are always rendered simultaneously

Source: https://kenney.nl/assets/racing-pack

"BCH" Example Exercise

We are only considering foreground objects (not the racetrack/grass) here for clarity (we would have to draw way more circles otherwise)

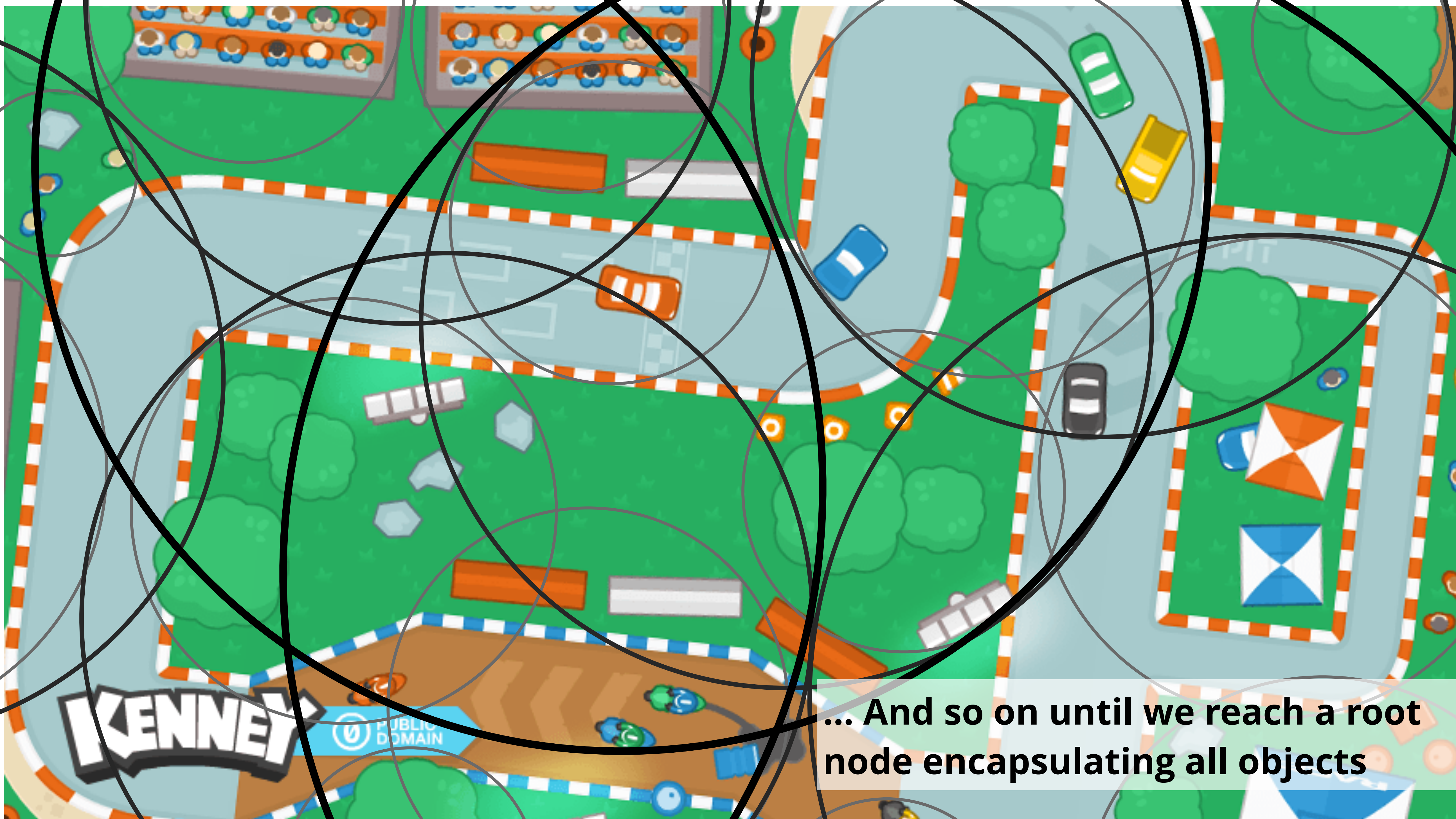


This is one level higher than the leaf nodes. Notice that there are exactly two sub-volumes for each parent volume and that volumes can and do overlap

Source: https://kenney.nl/assets/racing-pack

"BCH" Example Exercise

We are only considering foreground objects (not the racetrack/grass) here for clarity (we would have to draw way more circles otherwise)



Source: https://kenney.nl/assets/racing-pack

“BCH” Implementation

- Here’s how we could write the header file for this rendering engine.
- “Flattened Tree” arrangement in the primitives vector
  - start and range are valid for parent nodes and leaf nodes (same in Scotty3D!)

```
/**
 * A node in the BVH accelerator aggregate.
 * The accelerator uses a "flat tree" structure where all the primitives are
 * stored in one vector. A node in the data structure stores only the starting
 * index and the number of primitives in the node and uses this information to
 * index into the primitive vector for actual data. In this implementation all
 * primitives (index + range) are stored on leaf nodes. A leaf node has no child
 * node and its range should be no greater than the maximum leaf size used when
 * constructing the BVH.
 */
struct BVHNode {
  BVHNode(BBox bb, size_t start, size_t range)
      : bb(bb), start(start), range(range), l(NULL), r(NULL) {}
```

↳ **From bvh.h in Scotty3d**

```
class Primitive {
    // ...
    Color color_at_pixel(float x, float y);
    // ...
};

struct Circle {
    float x;
    float y;
    float rad;

    inline bool isect(float x_, float y_) {
        return (x_-x)*(x_-x)+(y_-y)*(y_-y) < rad*rad;
    }
};

struct BCHNode {
    BCHNode *l;
    BCHNode *r;

    Circle bounds;
    size_t start;  // start index in Scene::primitives
    size_t range;  // number of elts in primitives
};

struct BCHAccel {
    // ...
    BCHNode *root;
};

class Scene {
    // ...
    std::vector<Primitive *> primitives;
    BCHAccel *accel;

    Color color_at_pixel(float x, float y);
};
```

“BCH” Implementation

- Basic implementation of color_at_pixel(…) without acceleration structure. Instead of drawing to every bordering pixel for each primitive (like DrawSVG), we render every primitive at each pixel.

```
Color Scene::color_at_pixel(float x, float y) {
    Color cur = Color(0,0,0,0);
    for(int i = 0; i < primitives.size(); ++i) {
        Color top = primitives[i].color_at_pixel(x, y);
        // Alpha blend "over" operator (like DrawSVG)
        cur = Color::over(cur, top);
    }
    return cur;
}
```

- How would we traverse the BCH, given this header?

```
class Primitive {
    // ...
    Color color_at_pixel(float x, float y);
    // ...
};

struct Circle {
    float x;
    float y;
    float rad;

    inline bool isect(float x_, float y_) {
        return (x_-x)*(x_-x)+(y_-y)*(y_-y) < rad*rad;
    }
};

struct BCHNode {
    BCHNode *l;
    BCHNode *r;

    Circle bounds;
    size_t start;  // start index in Scene::primitives
    size_t range;  // number of elts in primitives
};

struct BCHAccel {
    // ...
    BCHNode *root;
};

class Scene {
    // ...
    std::vector<Primitive *> primitives;
    BCHAccel *accel;

    Color color_at_pixel(float x, float y);
};
```

“BCH” Implementation

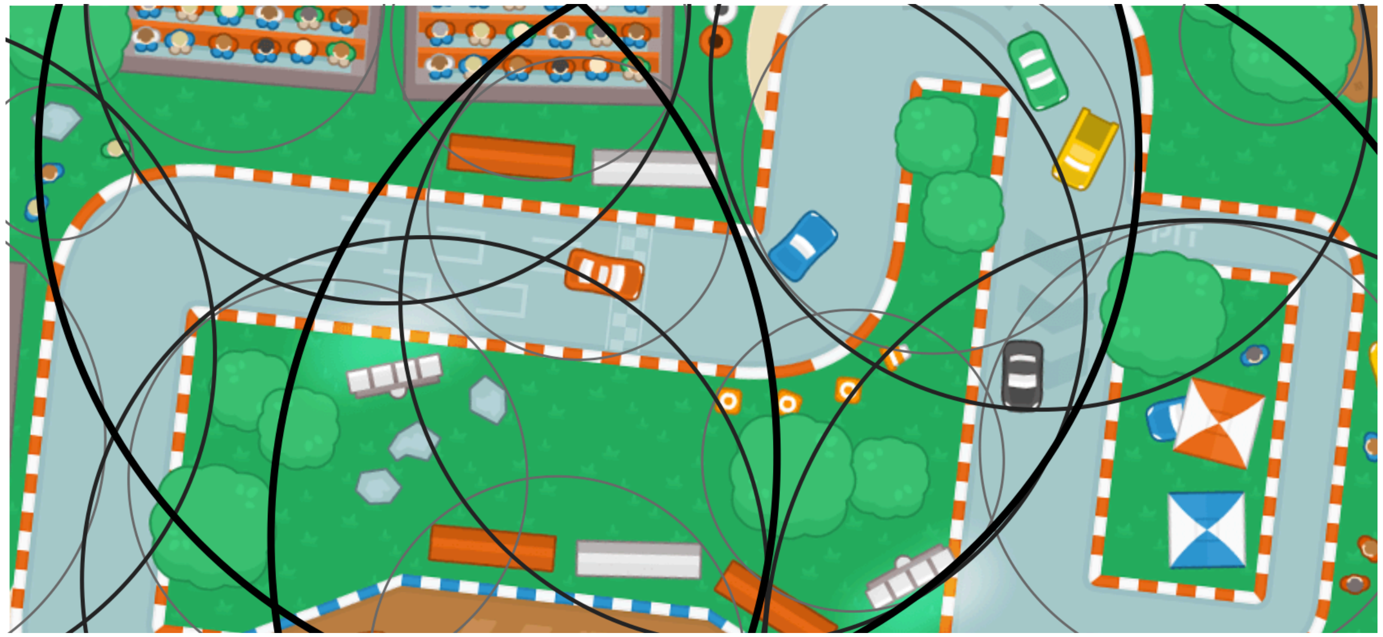
- Implementation of color_at_node that traverses a BCH

```
void Scene::color_at_node(float x, float y,
                          BCHNode *node, Color *cur) {
    if(!node->bounds.isect(x, y)) return;

    if(node->l == nullptr && node->r == nullptr) { // leaf
        for(int i = 0; i < node->range; ++i){
            int j = node->start + i;
            Color top = primitives[j].color_at_pixel(x, y);
            *cur = Color::over(*cur, top);
        }
        return;
    }

    color_at_node(x, y, node->l, cur);
    color_at_node(x, y, node->r, cur);
}

Color Scene::color_at_pixel(float x, float y) {
    Color ret(0,0,0,0);
    color_at_node(x, y, root, &ret);
    return ret;
}
```

- Sidebar: While faster, this method subtly changes (breaks) alpha blending behavior —
  Why? *How do you fix it?*

```
class Primitive {
    // ...
    Color color_at_pixel(float x, float y);
    // ...
};

struct Circle {
    float x;
    float y;
    float rad;

    inline bool isect(float x_, float y_) {
        return (x_-x)*(x_-x)+(y_-y)*(y_-y) < rad*rad;
    }
};

struct BCHNode {
    BCHNode *l;
    BCHNode *r;

    Circle bounds;
    size_t start;   // start index in Scene::primitives
    size_t range;   // number of elts in primitives
};

struct BCHAccel {
    // ...
    BCHNode *root;
};

class Scene {
    // ...
    std::vector<Primitive *> primitives;
    BCHAccel *accel;

    Color color_at_pixel(float x, float y);
    void color_at_node(float x, float y,
                       BCHNode *node, Color *cur);
};
```

BCH Takeaways / Questions



- Our aim has been to minimize the number of primitives considered by our renderer at each pixel
- Is the BCH successful? How much wasted space (space in each bounding volume that is not covered by a primitive) is there? How much overlap between nodes is there?
  - Both of these issues lead to redundant BVH traversals
- Is a circle the best bounding shape for this scene?
- What types of scenes would a BCH be most effective for?

Back to 3D: Building a BVH

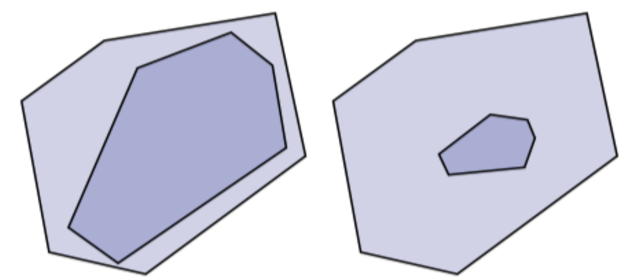
- With the BCH, we want to minimize "wasted space" in building the actual bounding partition.
  - We could theoretically find the best partitioning of a BCH by brute-forcing all possible partitions and maximizing the ratio of primitive areas to circle areas.
  - Try coming up with faster partitioning schemes…
- Now consider a pathtracer. What do we want to minimize in our acceleration structure to improve performance?
  - Recall, our query is ray / primitive intersections
  - Intersecting ray directions are totally unpredictable given the randomness of BRDFs
  - How do we "score" a particular partitioning?

The Surface Area Heuristic

Recall From previous lecture:

- **For convex object A inside convex object B, the probability that a random ray that hits B also hits A is given by the ratio of the surface areas S_A and S_B of these objects.**

$$P(\text{hit}A|\text{hit}B) = \frac{S_A}{S_B}$$



Leads to surface area heuristic (SAH):

$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

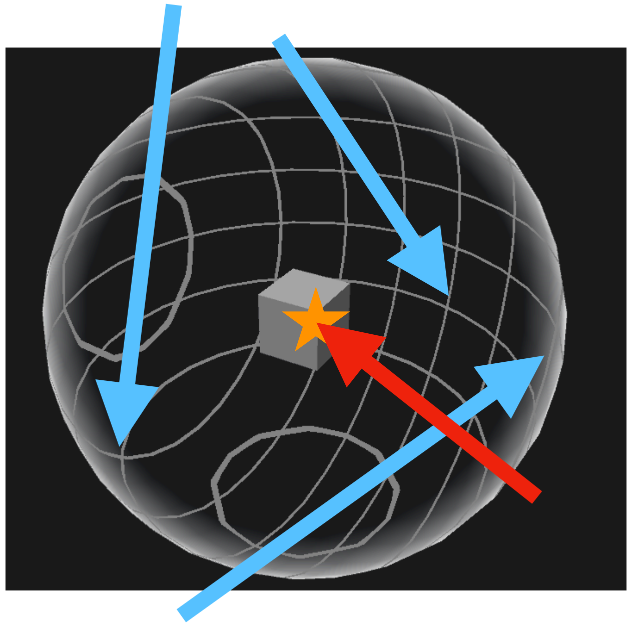
Assumptions of the SAH (*which may not hold in practice!*):

- **Rays are randomly distributed**
- **Rays are not occluded**

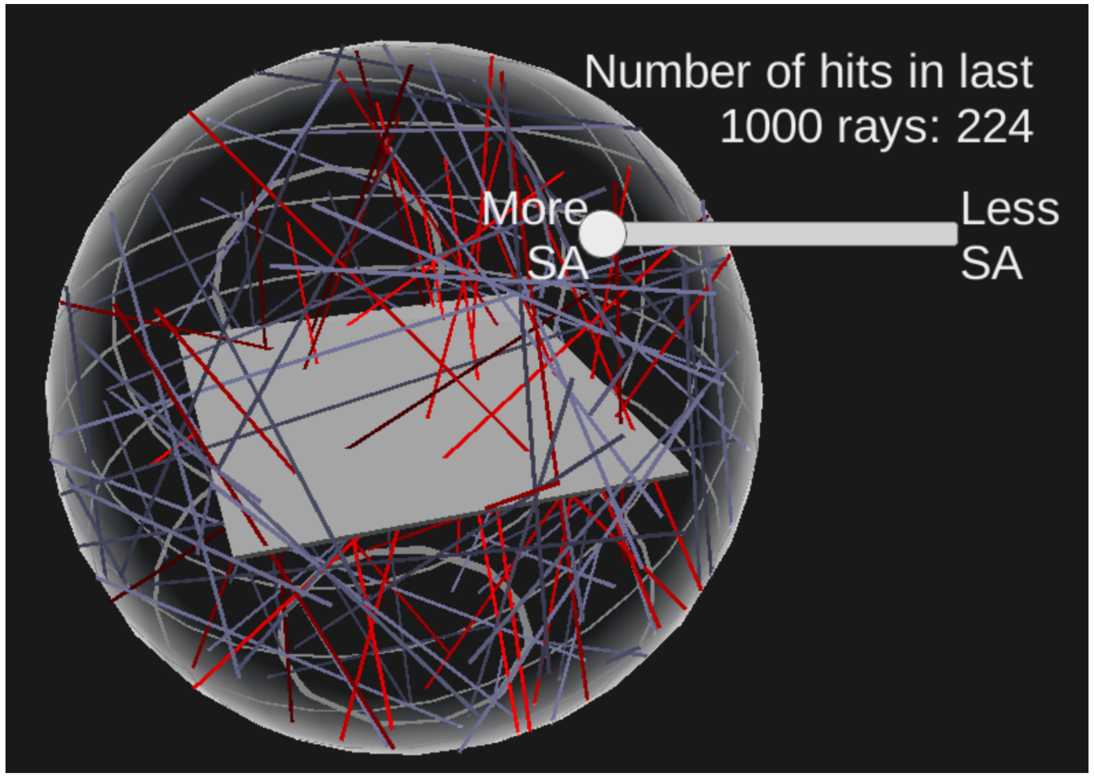
In short, the SAH is the way that we "score" a BVH, for the <u>specific application of ray-primitive intersection</u>. What kind of queries would the SAH be *bad* at? What alternatives to the SAH are there?

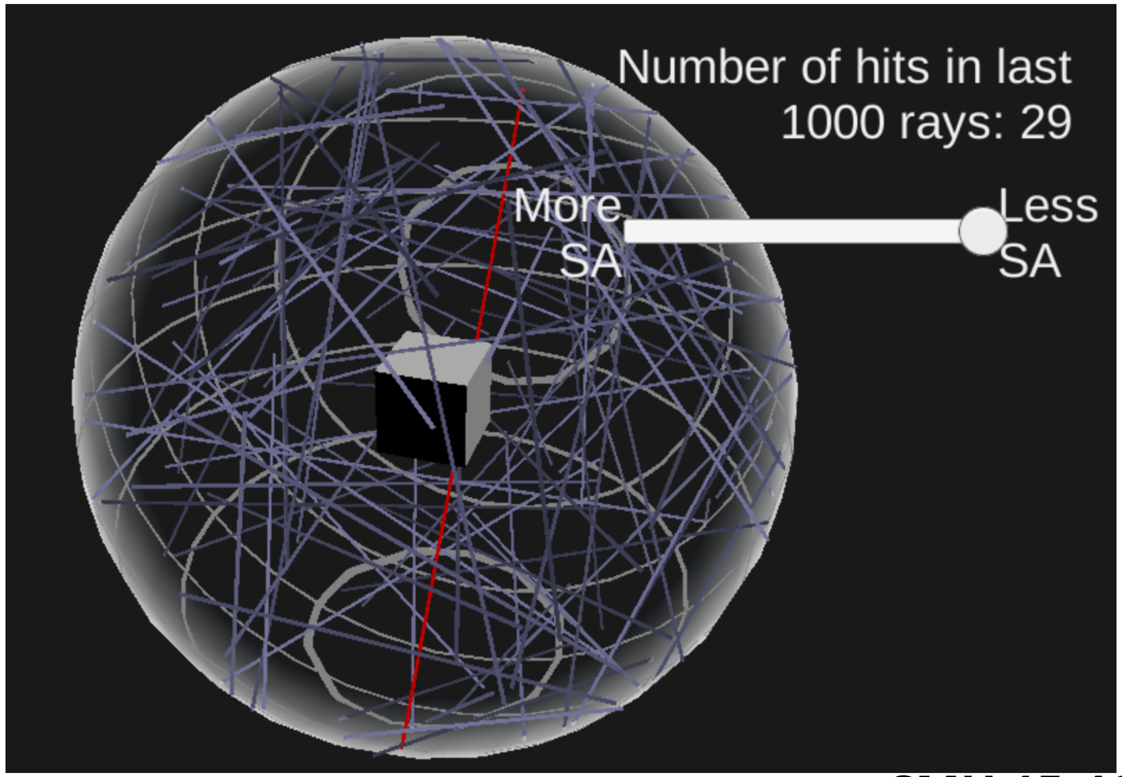
The Surface Area Heuristic

- ***Demo***: Squashing a cube while preserving its volume increases its Surface Area.
- How often should we expect to hit cubes with the same volume but different surface areas, with the sample rays randomly distributed about a sphere?



http://flafla2.github.io/demos/sah-vis/index.html





SAH in an axis-aligned BVH

- When building a BVH, we need to figure out how to partition the primitives, starting at the root node (partitioning into initial left/right subtree) and then recursively partitioning each subtree.
  - We use the SAH to choose our partitioning (minimize C)
  - Why do we need the $N_{\{A,B\}}$ term? The $\frac{S_{\{A,B\}}}{S_N}$ term?

 # primitives in subtree A

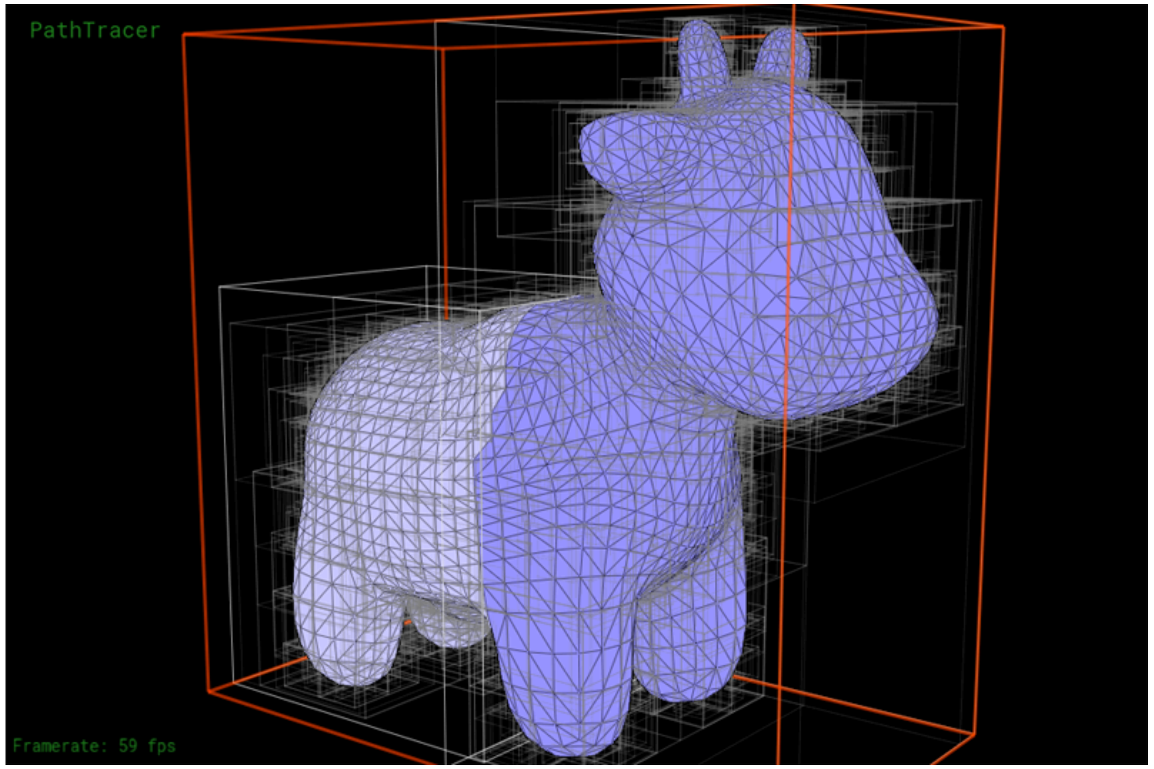
 # primitives in subtree B

SA of *bounding box of* subtree A

SA of *bounding box of* subtree B

$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

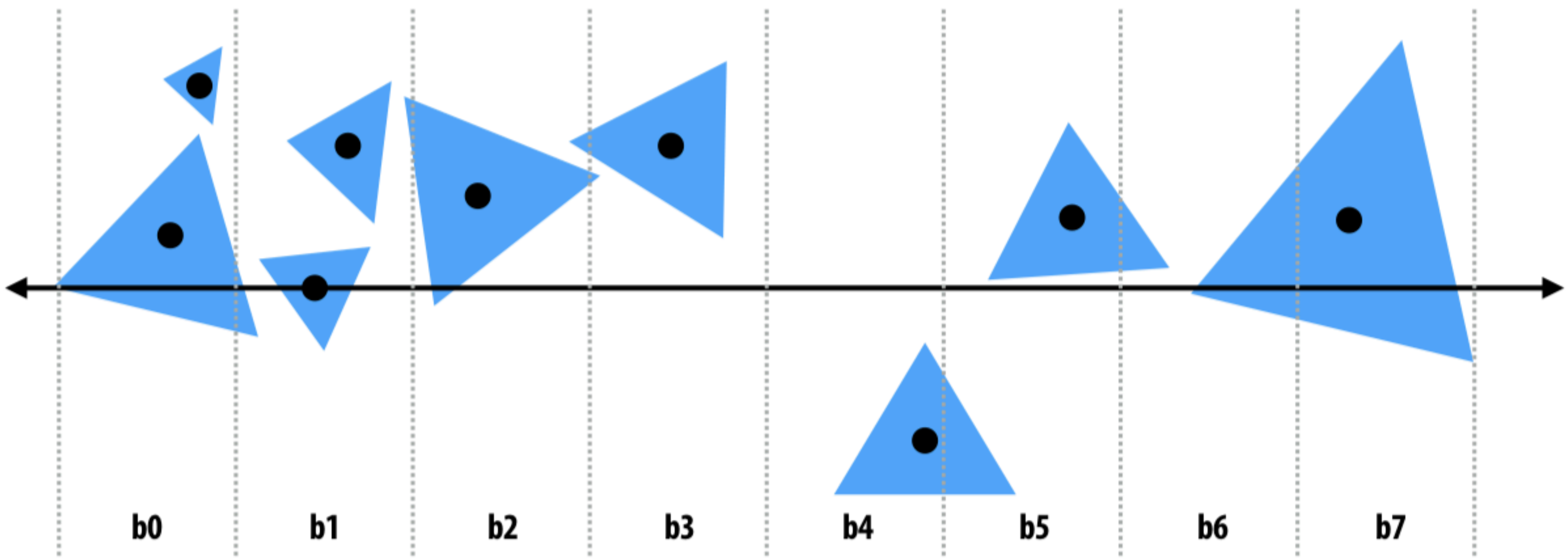
SA of *bounding box of* parent



- You can assume C_{trav} and C_{isect} are 1 as they are constants (irrelevant for comparisons)

Efficiently implementing partitioning

- **Efficient modern approximation: split spatial extent of primitives into B buckets (B is typically small: B < 32)**




```
For each axis: x,y,z:
  initialize buckets
  For each primitive p in node:
    b = compute_bucket(p.centroid)
    b.bbox.union(p.bbox);
    b.prim_count++;
  For each of the B-1 possible partitioning planes evaluate SAH
Recurse on lowest cost partition found (or make node a leaf)
```

(from previous lecture)

Questions?