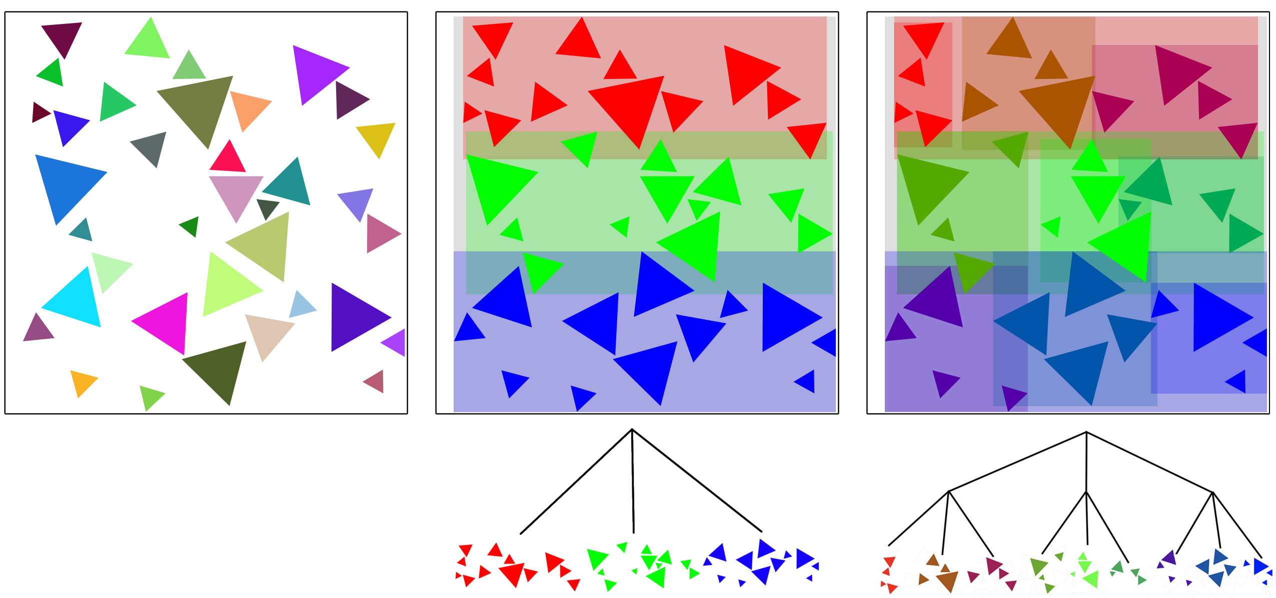
CSCI-1200 Data Structures — Fall 2019
Homework 8 — Bounding Volume Hierarchy Trees

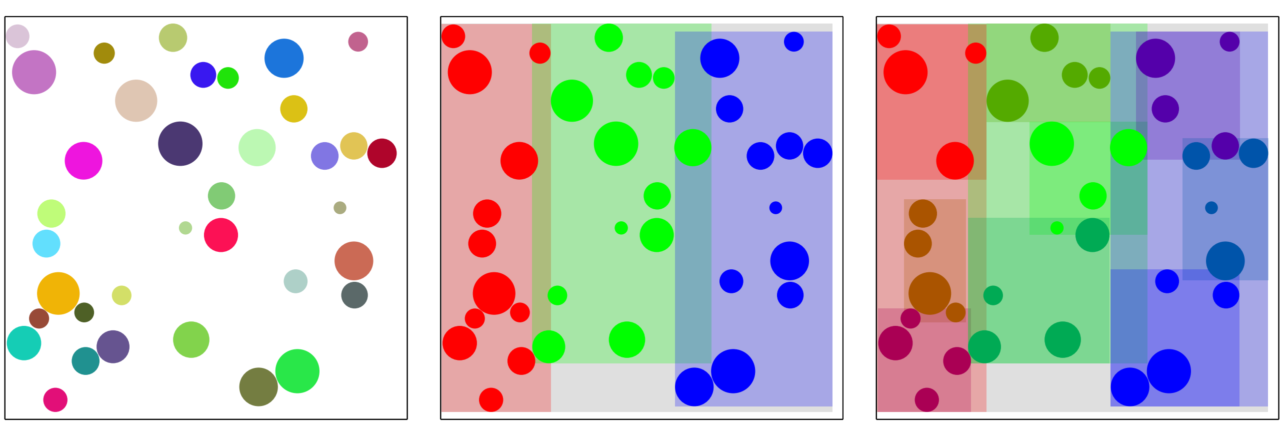
In this assignment you will build a templated spatial data structure called a *bounding volume hierarchy* (BVH) to store geometric objects such as disks, triangles, and quadrilaterals. A BVH facilitates efficient spatial queries for a variety of applications, including: ray tracing in computer graphics, collision detection for simulation and gaming, motion planning for robotics, nearest neighbor calculation, and image processing. Other spatial data structures with similar performance to bounding volume hierarchies for these tasks include *quad trees/octrees*, *k-d trees*, and *binary space partitions*.

Our `BVH` implementation will share some of the framework of the `ds_set` implementation we have seen in lecture and lab. You are encouraged to carefully study that implementation as you work on this homework.

The diagrams below illustrate a *ternary* two-dimensional BVH that was constructed from an input with 36 triangles on the left. In the middle we show the first level of the BVH tree (if we count the root node as "level 0"), and on the right we show the first and second levels of the BVH tree. Each `Node` of our tree data structure will always have 3 children (or no children if its a leaf node). Note: A general BVH can have any number of children (2 children is most common). Our data structure will only support 2D data, but bounding volume hierarchies can be implemented for 3D or higher-dimensional data.



Every node in the BVH stores the combined bounding box of all of the data stored in that subtree of the BVH. The individual shapes are only stored in the leaf nodes. Every shape is only stored in one leaf node! And here's another small example dataset using disks:

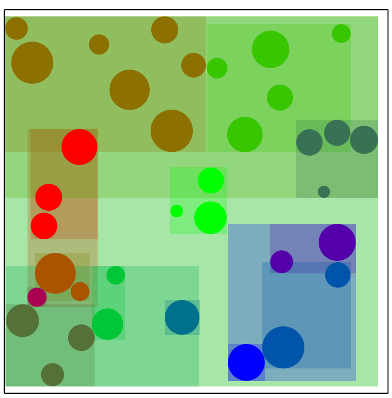
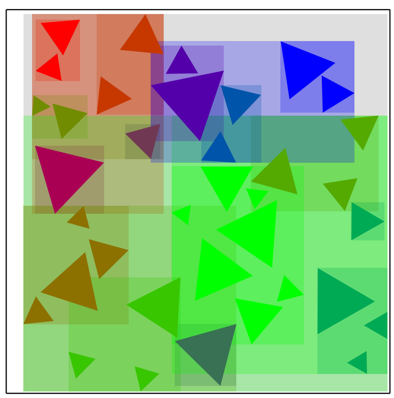
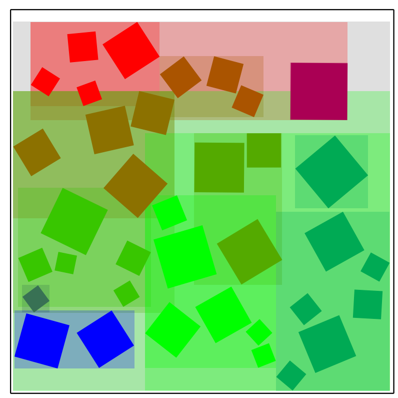


The algorithm to construct a BVH from a collection of geometry is to split the objects into 3 groups, compute the *bounding box* (minimum and maximum coordinates) of each group, and recurse. Note in the middle image for each example above we have colored the triangles by their group for the first split (red, green, and blue), and we've highlighted the bounding box of each group with a transparent box of the corresponding color. We will stop splitting and recursing when either the number of elements in a node is less than or equal to k (in this example, k is 4) or when we reach our specified limit on the depth or height of the longest path from root to leaf (in this example the maximum depth is 2).

The goal of the BVH is to cluster objects that are spatially close together and to minimize the surface area or volume of the bounding boxes for the elements in each node. Thus a good method for separating the objects into groups is to sort them by one of the axes (whichever dimension is longer) and then form c=3 groups with approximately the same number of elements. We sort the triangles by their centroid.

Note that the bounding boxes do not necessarily cover the entire screen and the boxes for the different groups will often overlap! If we instead assigned geometric objects to these three groups randomly, the calculated bounding boxes would much more significantly overlap – so let's not do that!

Alternatively, we can construct a BVH *incrementally* by `insert`ing objects one at a time. IMPORTANT NOTE: A BVH constructed incrementally may be negatively impacted by the data ordering and the resultant tree is much less likely to be balanced. The diagrams below show trees formed incrementally, but limited to 3 levels beyond the root node. The result depends on the insertion order and the specific algorithm for incremental insertion (there are multiple good strategies).

As mentioned earlier, one important application of spatial data structures is intersection or collision detection. Since each node of the tree knows the combined bounding box of all elements in that subtree, if our test object does not intersect with the combined bounding box of one of the children, we know the test object does not intersect or collide with any object in that subtree. Without a spatial data structure, intersection testing for a large group of n objects will be $O(n^2)$. With a good spatial data structure that running time is expected to be $O(n \log n)$.

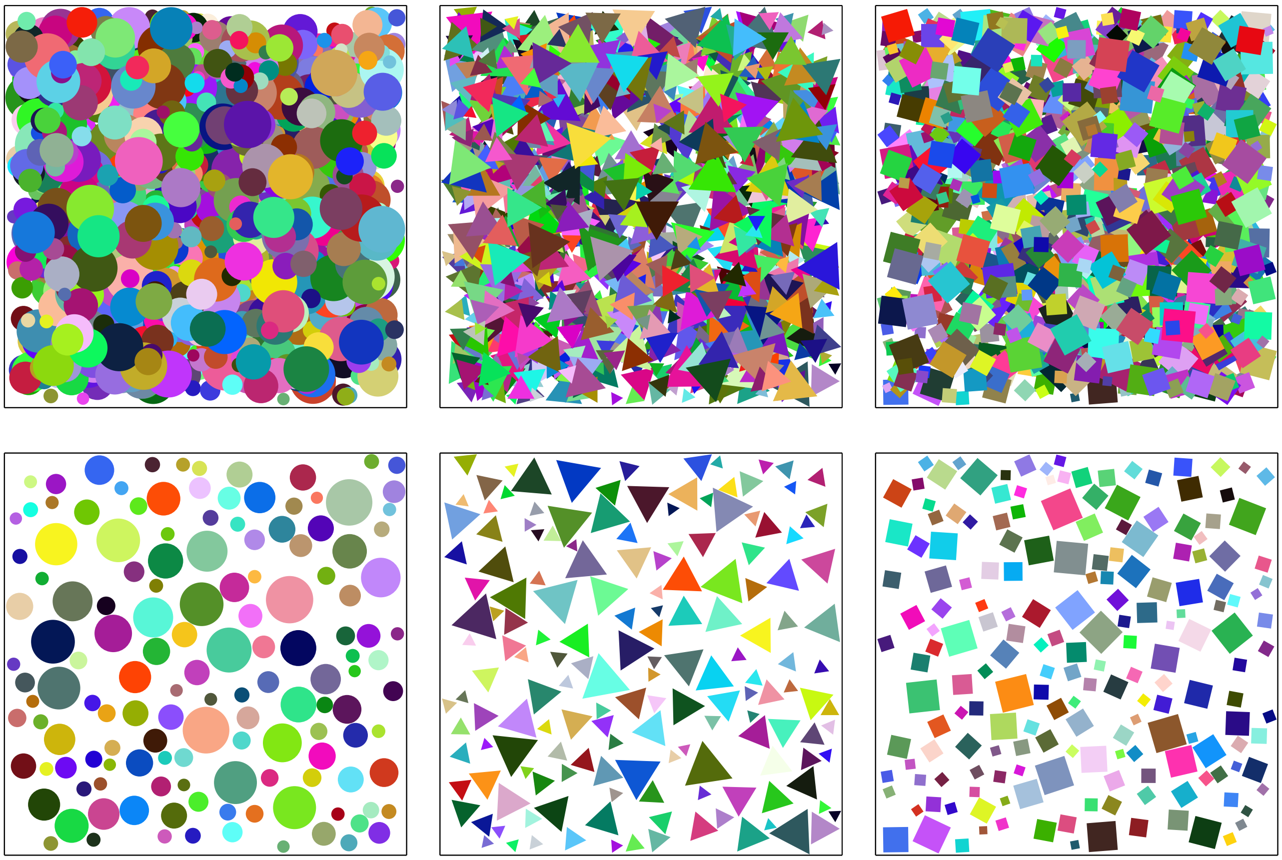
So how do we know what's "good" for our spatial data structure? And how do we compare different choices? In "Incremental BVH construction for ray tracing" (Computers & Graphics 2014) Bittner, Hapala, and Havran summarize the *surface area heuristic (SAH)*:

$$cost = \frac{1}{Area(scene)} \left[\sum_{I \,\in\, innernodes} Area(I) + \sum_{L \,\in\, leafnodes} Area(L) * ElementCount(L) \right]$$

They calculate the sum of the (surface) area of all inner (non-leaf) nodes and divide by the area of the entire scene. It is generally better for the inner nodes to have small surface area – which indicates the hierarchy is small and efficient, and we can quickly navigate to the relevant leaf nodes. They also calculate the sum of the leaf node surface area (again divided by the scene area). The leaf node area is multiplied (penalized) by the number of elements at that leaf node. We hope this number is a small constant.

Application: Poisson Disk Sampling

To test the efficiency of your BVH, we'll stress test the code with an application for generating random samples on a surface that are nicely packed, but separated by a minimum distance (or in our case, just guaranteed not to overlap). The input will be a large collection of geometric objects that overlap (see the top row below). Your task is to add these objects to your BVH one at a time, but *only if they do not intersect with any object already in the structure*. Sample output is in the second row below. Because we are all working with the same input dataset and using the same intersection code we *should* get the exact same result. (However, floating point comparisons can be unreliable due to rounding error.)



Implementation

Your task for this homework is to implement the templated `BVH` data structure. We recommend that you begin by carefully studying the provided code, which includes the implementation of the `Disk`, `Triangle`, and `Quad` classes. The code also includes key helper functions to generate *scalable vector graphics* (SVG) visualizations and evaluate the quality heuristic equation above. We also provide code to compare geometric objects and determine if they intersect or collide.

Note: The Disk, Triangle, and Quad classes have a lot of similarity and it would be quite natural (and a good design) to instead implement them using a common base class and C++ class inheritance. This would allow us also to mix the different shapes within a single example. We will cover inheritance in C++ later in the term. But for this assignment we are choosing instead an implementation with templated classes.

You will need to create a BVH in two ways, given all of the data at once in a vector, and incrementally, by inserting data one element at a time. You will also need to implement the iterator increment function: `operator++`. Once you have the basics working, clean up your class and test both pre- and post- increment and decrement operations. As this is a class with dynamically-allocated memory, you will also need to implement, test, and debug the copy constructor, assignment operator, and destructor. The homework server will compile and run your `bvh.h` file with the instructor's solution to test your implementation of these functions. It will test your code with Dr. Memory and your program must be memory error and memory leak free for full credit.

Command Line Arguments & Detailed Output

Study the `main.cpp` file and the available command line options. The examples in this handout were produced with the following commands. The zip file contains many other input examples with a variety of sizes of data to help you test the performance of your data structure.

```
./bvh.out -i 36.triangles -o 36_triangles.html
./bvh.out -i 36.triangles -o 36_triangles_1.html --depth_limit 1 --visualize
./bvh.out -i 36.triangles -o 36_triangles_2.html --depth_limit 2 --visualize

./bvh.out -i 36.disks -o 36_disks.html
./bvh.out -i 36.disks -o 36_disks_1.html --depth_limit 1 --visualize
./bvh.out -i 36.disks -o 36_disks_2.html --depth_limit 2 --visualize

./bvh.out -i 36.disks     -o 36_disks_incremental_3.html     --depth_limit 3 --incremental --visualize
./bvh.out -i 36.triangles -o 36_triangles_incremental_3.html --depth_limit 3 --incremental --visualize
./bvh.out -i 36.quads     -o 36_quads_incremental_3.html     --depth_limit 3 --incremental --visualize

./bvh.out -i overlap_1000.disks     -o overlap_1000_disks.html
./bvh.out -i overlap_1000.triangles -o overlap_1000_triangles.html
./bvh.out -i overlap_1000.quads     -o overlap_1000_quads.html

./bvh.out -i overlap_1000.disks     -o discard_overlap_1000_disks.html     --depth_limit 8 --incremental --discard_overlap
./bvh.out -i overlap_1000.triangles -o discard_overlap_1000_triangles.html --depth_limit 8 --incremental --discard_overlap
./bvh.out -i overlap_1000.quads     -o discard_overlap_1000_quads.html     --depth_limit 8 --incremental --discard_overlap
```

Each of these commands produce a Scalable Vector Graphics (SVG) file stored as a `.html` file. You should be able to open this file locally in a standard web browser (e.g., Chrome, Firefox, etc.) We will be autograding by looking at the STDOUT from each run. For example, this command:

```
./bvh.out -i 36.triangles -o 36_triangles_2.html --depth_limit 2 --visualize --print_tree > 36_triangles_2_stdout.txt
```

Will produce the output below:

```
inner area=0.929 [< 45.7  12.9> <992.3 993.9>]
   inner area=0.319 [< 68.6  12.9> <969.5 367.5>]
      leaf area=0.045 [< 68.6  27.3> <212.6 337.1>] elements: [< 68.6 224.1> <116.3 277.0>] [< 77.6 115.0> <145.0 186.5>] [< 90.5  27.3> <192.7 120.0>] [<121.1 245.2> <212.6 337.1>]
      leaf area=0.109 [<236.4  12.9> <567.0 343.1>] elements: [<236.4 174.6> <327.6 273.9>] [<296.7  12.9> <410.2 116.7>] [<416.0  94.6> <500.0 168.0>] [<377.0 159.6> <567.0 343.1>]
      leaf area=0.117 [<558.9  83.6> <969.5 367.5>] elements: [<558.9 197.8> <663.7 301.1>] [<715.1  83.6> <857.0 234.4>] [<821.4 172.3> <906.6 268.9>] [<870.5 276.6> <969.5 367.5>]
   inner area=0.367 [< 76.0 298.5> <984.8 701.9>]
      leaf area=0.131 [< 76.0 298.5> <401.2 701.9>] elements: [< 76.0 355.0> <255.2 532.0>] [<158.4 511.3> <217.8 570.9>] [<215.7 598.1> <319.1 701.9>] [<310.3 298.5> <401.2 389.6>]
      leaf area=0.097 [<431.3 317.2> <705.4 672.5>] elements: [<431.3 508.7> <481.0 561.9>] [<507.8 317.2> <598.9 398.9>] [<506.4 409.1> <642.6 527.1>] [<546.6 496.6> <705.4 672.5>]
      leaf area=0.087 [<624.4 360.4> <984.8 600.8>] elements: [<624.4 466.0> <682.7 520.1>] [<637.6 360.4> <758.3 481.6>] [<824.3 439.4> <914.6 524.8>] [<898.8 502.0> <984.8 600.8>]
   inner area=0.377 [< 45.7 595.5> <992.3 993.9>]
      leaf area=0.128 [< 45.7 631.8> <399.2 993.9>] elements: [< 45.7 746.9> <124.2 818.8>] [< 90.4 631.8> <238.8 784.4>] [<163.1 890.9> <233.1 960.7>] [<335.0 928.7> <399.2 993.9>]
      leaf area=0.157 [<313.5 595.5> <721.0 979.7>] elements: [<313.5 697.1> <453.8 854.2>] [<439.5 818.8> <599.6 979.7>] [<491.6 595.5> <642.2 758.4>] [<595.8 752.0> <721.0 871.6>]
      leaf area=0.079 [<704.9 673.3> <992.3 948.2>] elements: [<704.9 691.3> <775.0 761.2>] [<810.6 673.3> <960.0 845.3>] [<887.4 889.0> <939.2 948.2>] [<931.2 787.1> <992.3 857.6>]
node count:          13
max depth:            2
inner node area:   1.992
leaf node area:    3.797
```

Extra Credit: Tree Balancing & Element Erase

For extra credit you can explore and evaluate alternate methods of constructing and/or incrementally inserting data to a BVH. Does one method consistently out-perform another? Either measured timing results or through the surface area heuristic?

We are not requiring the implementation of `erase` for your BVH. Why not? Discuss the implications of implementing and using erase on a BVH. How does it compare to erase on a standard binary search tree? Optionally, design, implement, test, and evaluate the performance of the erase operation on a BVH.

Homework Submission

Use good coding style and detailed comments when you design and implement your program. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your `README.txt` file.