CSCI 1933 Project 5

Bounding Volume Hierarchy

Due Date: Friday, May 4th, 2018 before 11:55pm

Instructions

Please read and understand these expectations thoroughly. Failure to follow these instructions could negatively impact your grade. Rules detailed in the course syllabus also apply but will not necessarily be repeated here.

- **Due:** The project is due on **Friday, May 4th** by **11:55 PM**.
- **Identification:** Place you and your partner's x500 in a comment in all files you submit. For example, `//Written by over0219 and brow2327`.
- **Submission:** Submit a `zip` or `tar` archive on Moodle containing all your `java` files.

  You are allowed to change or modify your submission, so submit early and often, and *verify that all your files are in the submission.*

  Failure to submit the correct files will result in a score of zero for all missing parts. Submissions with or incorrect format (e.g., `.rar`, `.7z`, `.java`) will be penalized. Only submissions made via Moodle are acceptable.
- **Partners:** You may work alone or with *one* partner. **Failure to tell us who is your partner is indistinguishable from cheating and you will both receive a zero**. Ensure all code shared with your partner is private.
- **Code:** You must use the *exact* class and method signatures we ask for.

  We use unit tests to evaluate your code. Code that doesn't compile will receive a significant penalty. Code should be compatible with Java 8, which is installed on the CSE Labs computers.
- **Questions:** Questions related to the project can be discussed on Moodle in abstract. **Do not post any code or solutions on the forum**.
- **Grading:** Grading will be done by the TAs, so please address grading problems to them *privately.*

Code Style

Part of your grade will be decided based on the "code style" demonstrated by your programming. In general, all projects will involve a style component. This should not be intimidating, but it is fundamentally important. The following items represent "good" coding style:

- Use effective comments to document what important variables, functions, and sections of the code are for. In general, the TA should be able to understand your logic through the comments left in the code.

  Try to leave comments as you program, rather than adding them all in at the end. Comments should not feel like arbitrary busy work - they should be written assuming the reader is fluent in Java, yet has no idea how your program works or why you chose certain solutions.
- Use effective and standard indentation.
- Use descriptive names for variables. Use standard Java style for your names: `ClassName`, `functionName`, `variableName` for structures in your code, and `ClassName.java` for the file names.

Try to avoid the following stylistic problems:

- Missing or highly redundant, useless comments. `int a = 5; //Set a to be 5` is not helpful.
- Disorganized and messy files. Poor indentation of braces (`{` and `}`).
- Incoherent variable names without context.
- Slow functions. While some algorithms are more efficient than others, functions that are aggressively inefficient could be penalized even if they are otherwise correct.

  IMPORTANT: For this project, slow functions are indicative of an incorrect implementation.

The programming exercises detailed in the following pages will both be evaluated for code style. This will not be strict – for example, one bad indent or one subjective variable name are hardly a problem. However, if your code seems careless or confusing, or if no significant effort was made to document the code, then points will be deducted.

In further projects we will continue to expect a reasonable attempt at documentation and style as detailed in this section. If you are confused about the style guide, please talk with a TA.

Note: This is a brand new project. If you notice any errors or have suggestions for improvements, please let us know.

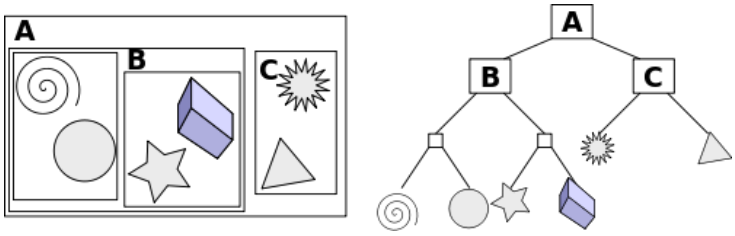
Bounding Volume Hierarchy

A bounding volume hierarchy (BVH) is an essential tool in computer graphics. It's used in rendering, collision detection, and much more. There are different kinds of acceleration structures, but the one you're required to implement is an Axis-Aligned Bounding Box (AABB) tree, with top-down construction.

The fun thing about acceleration structures is that they make use of several fundamental techniques in Computer Science, such as trees, recursion, queues/stacks, polymorphism, and more. This is a good opportunity to make use of the skills and knowledge you've acquired in this course in a (hopefully) fun and interesting final project.

AABB Tree

Geometric objects in space are grouped together in overlapping boxes. Boxes with one object in them are the leaf nodes of a tree. Larger boxes that enclose smaller ones are parent nodes. The image below (left) shows various shapes in world space being partioned into a tree (right). Although its leaf nodes have two objects each, it illustrates the general idea.
See this Wiki page for more details.



How can we use this tree? Let's say there was a screen open with the geometry of the above figure and the user wants to select the circle with a mouse by clicking on it. To see if the circle was selected, we start at the root node (A). Then, based on the coordinates of the mouse, we check which of its children the pointer is hovering over (B). We traverse to the (B) node, traverse its children (and so forth), until we reach a leaf and see if the click was in bounds of the shape.

Why not just loop over all of the geometry and test them directly? Consider what happens when we have one million shapes instead of six. What about ten million? Checking if the cursor is within the bounds of a box is a lot less expensive to compute than a 15-pointed star. Using a tree to process the selection of the click will cull millions of these expensive point-in-geometry tests, making the run time of your interface significantly faster.

This day and age, having to wait a few seconds to process a mouse click ends up with angry users and 1-star app ratings. We can't have that...

Objective

You are required to implement the primary function of building and traversing the AABB tree. Code that generates geometry, rendering, and handling mouse input is provided. The following files are on moodle:

- `AABBTree.java`: Contains the code for generating geometry and processing graphics. You won't have to change this file, but it might be helpful to look at and understand it.
- `Bounds.java`: The class for an axis-aligned bounding box. You'll need to implement this.
- `Circle.java`: The class for a cicle, containing geometry-intersection tests and rendering. You won't have to change this.
- `Node.java`: The class that makes up our tree! Most of the work you will be doing is here.
- `Shape.java`: A base class for geometry. You won't have to change this.
- `Vec2.java`: The class for points in space. You won't have to change this.

When running the program, a window should pop up containing a bunch of circles. Once you've finished your BVH implementation, clicking a circle should make it turn red. Clicking again will make it turn back to gray. For this project, you can assume that no shapes will overlap.

If your tree traversal is correct, this should run in real time. That is, you won't have to wait for the object to change colors.

Hint: It's highly recommended to test your methods with fewer geometry than the default. In `AABBTree.java` there is an `int max_shapes` variable in `main`. Start by setting this to one or two when implementing your tree for the first time.

1 Axis-Aligned Bounding Box

The first step is to implement the Axis-Aligned Bounding Box (AABB), found in `Bounds.java`. Be sure to fully test your methods. Methods marked with "TODO" are for you to complete. To get full points for this section you must implement

- `boolean isOutside(double x, double y)`: Returns true if the point (x, y) is outside the bounds of the box, false otherwise.
- `void extend(double x, double y)`: Grows the box (increases max, decreases min) to enclose the point.
- `void extend(Bounds b)`: Grows the box to enclose another box.
- `double exteriorDistance(double x, double y)`: Distance between the AABB's surface and a point in space. If the point is inside the bounds, return 0.

2 Node Constructor

```
Node(Stack<Shape> stack, int axis)
```

The node constructor takes a stack of shapes, a splitting plane axis, and initializes itself. The splitting plane axis is a number (0 or 1) that describes what axis on which we will do the partitioning. See §3 for more details.

To get full points for this section, implement the following:

- Extend the AABB to contain all shapes in the stack.
- If stack size is 1, become a leaf node.
- If stack size > 1, partition the stack using the `splitStack` method, increment the axis, and create children recursively.

3 Splitting the Stack

```
void splitStack(Stack<Shape> stack, int axis,
Stack<Shape> leftStack, Stack<Shape> rightStack)
```

The role of this method is to partition the stack into two new stacks (though not necessarily in half). The metric we're using to decide the partion is the **spatial median**. This is simply the average centroid of all objects in the stack (for a given axis).

For example, let's say the splitting axis was 0 (x-axis) and you had two shapes, centered at (1,2) and (2,3). The spatial median is 1.5. And so, the former shape (1,2) goes on the left stack, and the latter (2,3) on the right.

Do **not** create new shapes with the same location/radius. Make sure you are moving around the same shape object since their references are used internally by the renderer.

To get full points for this section, you must:

- Compute the spatial median.
- Partition the stack into leftStack and rightStack based on the spatial median.

4 Shape Selection

```
boolean select(double x, double y, int[] counter)
```

On a left-click, the `select` method is used to traverse the tree and identify the shape whose boundary includes the point (x,y). It returns true if an object was selected, false otherwise.

Here you will take advantage of the tree data structure to accelerate the search. That is, if the point (x,y) is outside the bounds of the node's AABB, the point is *also* outside the bounds of children nodes, thus we can simply return false. Otherwise, continue searching down the tree.

To get full points for this section, implement the following:

- Early exiting if the point (x,y) is outside the bounds of the node.
- If on a leaf, check to see if (x,y) is physically within the bounds of the shape.
- Recursive traversal of the left and right children.

5 Honors

```
boolean nearest(double x, double y,
double[] currentMin, Shape[] shapeRef, int[] counter)
```

The `nearest` method is very similar to the `select` method from §4. Instead of selecting a shape when the (x,y) coordinate is above it, however, `nearest` will select the *nearest* shape with respect to the input point. It returns true if it found a shape closer to the point (x,y), false otherwise. This happens on a right-click.

There are two more arguments to set. Note that both currentMin and shapeRef are single-element arrays. When you've found a candidate nearest object, in that its exterior distance to the point is the current minimum, set both `currentMin[0]` and `shapeRef[0]`. The former is that exterior distance, while the latter is a reference to shape itself. Remember that exterior distance is zero if the point is inside the shape.

The other major difference between `nearest` and `select` is that you should traverse the node that is closer to the point (x,y) first. This is because it's more likely to contain the actual nearest shape, and possibly cull future traversals!

To get full points for this section, implement the following:

- Early exiting if the node's boundary is further away than the candidate nearest shape.
- If on a leaf, check to see if the node holds a closer shape. If so, set the necessary references.
- Recursive traversal of the left and right children.