Improved Collision Detection in StarLogo Nova

by Divya Bajekal

S.B., C.S. MIT, 2014

S.B., Math MIT, 2014

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2015

© Massachusetts Institute of Technology 2015. All rights reserved.
Improved Collision Detection in StarLogo Nova
by Divya Bajekal
Submitted to the Department of Electrical Engineering and Computer Science

May 22, 2015
In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT
StarLogo Nova is blocks-based educational software that allows students to write and play their own 3D games online. It is the online version of StarLogo TNG. This thesis explores the problem of needing more accurate collision detection in StarLogo Nova while maintaining reasonable performance. Three new collision detection systems for StarLogo Nova are developed and evaluated. Compared to the spheres used to perform collision checks in the current system, the first new system, called the TightestFitCollider, introduces a variety of bounding spheres, bounding boxes, and bounding capsules as bounding structures that may fit the models in StarLogo Nova more closely. The second system, called the HierarchicalCollider, uses hierarchies of bounding boxes to perform even more precise collision detection than the TightestFitCollider. Finally, the third system combines the first two systems, so that the advantages of each can be used as appropriate. The three systems are evaluated for their accuracy and performance within the StarLogo Nova framework.
Acknowledgements

I would like to thank my thesis advisor, Eric Klopfer, for his guidance, support, and feedback throughout my thesis project, as well as earlier in my UAP and UROP semesters. This thesis also would not have been possible without my mentor Daniel Wendel. Thank you, Daniel, for all of the knowledge about StarLogo Nova and computer graphics that you shared with me, the many hours you spent discussing and providing feedback for my work, and for making the lab a fun place where I looked forward to coming. I am also grateful to all of my friends who checked in regularly to see how I was doing and provided encouragement. Finally, I would like to thank my family for their wonderful moral support throughout my thesis.
## Contents

1 Introduction ................................................................................................................................. 8

1.1 Problem definition ...................................................................................................................... 9

1.2 Contributions ............................................................................................................................ 10

2 Background .................................................................................................................................. 12

2.1 StarLogo Nova .......................................................................................................................... 12

2.1.1 Programming interface ........................................................................................................ 12

2.1.2 Games engine ....................................................................................................................... 13

2.2 Current collision detection system ............................................................................................ 14

3 TightestFitCollider ....................................................................................................................... 17

3.1 Motivation .................................................................................................................................. 17

3.2 Design overview ......................................................................................................................... 18

3.3 Data structures .......................................................................................................................... 18

3.4 Finding tight fitting bounding structures .................................................................................. 20

3.4.1 Bounding spheres .................................................................................................................. 20

3.4.2 Bounding boxes ..................................................................................................................... 22

3.4.3 Bounding cylinders ............................................................................................................... 24

3.4.4 Bounding capsules ............................................................................................................... 25

3.5 Transforming bounding structures ........................................................................................... 27

3.6 Intersection tests ....................................................................................................................... 28

3.6.1 Feasibility of cylinder intersection tests ............................................................................... 28

3.6.2 Sphere-Sphere ...................................................................................................................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.2</td>
<td>Performance of finding bounding structures</td>
<td>55</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Performance of accessing and transforming bounding structures</td>
<td>56</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Performance of intersection tests</td>
<td>57</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Performance of overall system</td>
<td>58</td>
</tr>
<tr>
<td>6.3</td>
<td>HierarchicalCollider</td>
<td>59</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Performance of construction</td>
<td>59</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Performance of transformation</td>
<td>60</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Performance of traversal</td>
<td>60</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Performance of overall system</td>
<td>60</td>
</tr>
<tr>
<td>6.4</td>
<td>CombinationCollider</td>
<td>61</td>
</tr>
<tr>
<td>7</td>
<td>Future work</td>
<td>62</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>References</td>
<td>65</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2-1: Blocks-based interface in StarLogo Nova. ......................................................... 13
Figure 2-2: Collision detection in a paintball game. .............................................................. 14
Figure 2-3: Fireman and dragon model compared to current collision sphere. .......... 16
Figure 3-1: The fireman model with its bounding box (blue) and various bounding
capsules (red).......................................................................................................................... 27
Figure 3-2: Spheres with the distance between their centers shown in red. ............ 29
Figure 3-3: Two oriented boxes projected to a separating axis. ................................. 33
Figure 3-4: Two boxes that are almost colliding edge-to-edge........................................ 35
Figure 3-5: Sphere-Capsule intersection tests. ................................................................. 36
Figure 3-6: Sphere-Box intersection tests........................................................................ 38
Figure 3-7: A capsule projected to a separating axis......................................................... 39
Figure 4-1: Dragon model with its tightest fitting bounding box (blue). ................. 43
Figure 4-2: The first three levels of the dragon model hierarchy...................................... 47
Figure 6-1: Volumes of bounding structures for 12 representative models.......... 55
Figure 6-2: Twelve models, each shown with its number of vertices and time taken
to load the tightest fitting bounding structure................................................................. 56
Figure 6-3: Five representative models, each with the time taken to access and
transform the tightest fit bounding structure............................................................... 57
Figure 6-4: Performance of pairwise intersection tests..................................................... 58
Figure 6-5: Twelve models and the time taken to construct their bounding box
hierarchies............................................................................................................................... 60
1 Introduction

StarLogo Nova is a project that enables students to write and play their own 3D computer games online. The aim of StarLogo Nova is to help teach computer science to middle and high school students in a fun, intuitive, and interactive way. One major component of StarLogo Nova is its 3D graphics engine. The graphics engine has many responsibilities, such as rendering the 3D worlds accurately and modeling the physics of those worlds convincingly.

In order to model the 3D worlds convincingly, the graphics engine must detect collisions between characters and objects in the game quickly and accurately. As the characters and objects in the game move around, it is important to know which ones have collided with each other, so that they can behave realistically after the collision. For example, paintballs should only color a character when they hit that character. A human or animal character should not be able to walk through walls.

Collision detection is the problem of determining which of the many agents in a virtual scene have collided. Unfortunately, collision detection can be computationally intensive. Thus there is a tradeoff between accuracy and performance. In the current collision detection system, all characters and objects, known collectively as agents, are approximated as unit spheres that are scaled according to the agent’s size. Thus if the two spheres are colliding, the system returns that the two agents are colliding. Although it is quite fast to check if two spheres are colliding, spheres don’t approximate most agents that well. This means
that there are false positives when the spheres collide but the agents don’t actually collide, or false negatives when the spheres don’t collide but the agents do collide. This system is quite efficient, but not as accurate as users would like it to be. The purpose of this thesis is to develop alternate collision detection systems that are more accurate while maintaining reasonable performance.

1.1 Problem definition

This thesis seeks to improve collision detection within the context of StarLogo Nova. There are several constraints that differentiate this problem from collision detection in the general case. The first specific constraint is the choice of language. Collision detection for StarLogo Nova is written in ActionScript3. This language was chosen because StarLogo Nova is an online project, and when it was created, ActionScript3 was the only language with broad-based support for 3D games in a browser. Although ActionScript3 provides broad support, it also has some drawbacks that make collision detection more challenging. For example, ActionScript3 has limited built-in support for matrices and matrix operations. It also lacks built-in support for numerical equation solving. A second constraint is the types of models that are collided in StarLogo Nova. Common models include people, animals, nature (trees, shrubs, other plant life), and man-made models (spaceships, fortresses, etc.). Thus a variety of models must be taken into account. Finally, a third constraint is the user expectations. As mentioned above, users would like more accurate collision detection. In particular, collisions of main characters in games should be especially accurate, since most of the game-player’s attention is focused on that character.
Increased accuracy is expected to lead to a performance slowdown. However, the overall system should still be fast enough so that the frame rate is at least 25 frames per second. If the games play slower than that, the games will appear choppy, robotic, and unrealistic.

Thus there are two types of criteria I evaluate against for the collision detection systems developed in this thesis. First, for collision accuracy, I examine the volumes of the bounding shapes that are used to approximate the agents in the collision checks. These volumes reflect how tightly the bounding shapes fit the agents. Second, for collision detection performance, I measure the speeds of various algorithms used in both the pre-computation stage and in the gameplay stage of StarLogo Nova. I also measure the overall time collision checking takes for a sample game in StarLogo Nova for each of the collision detection systems. These times are compared to the time for the current collision detection system. The goal is to develop a collision detection system with improved accuracy that still provides fast enough performance for games to run smoothly in StarLogo Nova.

1.2 Contributions

I first designed two collision detection systems based on the constraints and requirements for StarLogo Nova. The first, the TightestFitCollider, is discussed in section 3. The second, the HierarchicalCollider, is discussed in section 4. Then I synthesized and built upon existing algorithms for finding tight-fitting bounding structures, computing pairwise intersection tests, and constructing and traversing hierarchies. I used these algorithms to implement the two collision detection
systems, taking care to write tests for correctness and robustness. I also left many parameters configurable for future fine-tuning and flexibility. Because of the tradeoffs between the two collision systems, I also combined them into a third system that is more flexible and reflects the best of both systems. This combination system is discussed in section 5. Finally, I evaluated the performance, both accuracy and speed, for the various collision detection systems. These evaluations are discussed in section 6. Before these contributions are discussed, the next section provides some background on StarLogo Nova and the current collision detection system.
2 Background

This section first describes StarLogo Nova in more detail, including its programming interface and its games engine. Then the current collision detection system is explained further.

2.1 StarLogo Nova

StarLogo Nova is meant to guide students in writing their own computer programs to create 3D computer games. Teachers also use StarLogo Nova to create educational simulations. There are two main components to this software: the programming interface and the 3D engine. Each of these plays an important role in making StarLogo Nova educational, fun, and easy to use.

2.1.1 Programming interface

StarLogo Nova uses a graphical interface in which different colored blocks represent the elements of the programming language. Students can fit these blocks together like puzzle pieces to develop the logic of their program, as shown in Figure 2-1. This blocks-based interface makes computer science and programming more accessible, especially to beginner students.
Figure 2-1: Blocks-based interface in StarLogo Nova. On the left is a drawer of blocks that can be dragged and dropped onto the right panel. On the right is a sample program that creates a turtle and assigns it certain attributes.

2.1.2 Games engine

The second major component of StarLogo Nova is its 3D engine. The 3D engine enables users to create rich and compelling 3D worlds for their games and simulations. The collision detection procedure is one component of the 3D engine. Figure 2-2 displays a paintball game created in StarLogo Nova. Collision detection is important in this game because the turtles are supposed to change colors when hit by a paintball. The capabilities of the 3D engine are important in making the software fun and exciting to use, which in turn is important in making StarLogo Nova an effective teaching tool.
Figure 2-2: Collision detection is an important component of this paintball game. The bear walks around trying to hit turtles with paintballs. When one of the turtles is hit, it changes color. The 3D engine is responsible for detecting these collisions.

### 2.2 Current collision detection system

The current collision detection system uses spheres to approximate all of the agent models. The vertices that describe each model are centered and scaled when that model is loaded. In particular, the vertices are centered at the origin in the x and y dimensions and scaled to fit within a unit cube in those two dimensions. In the z dimension, the model vertices may extend beyond the unit cube. The model is translated so that the entire shape is at or above the origin, and thus it is not centered in the z dimension. Then the spheres used in collision detection are centered at (0,0,0.5) and have a radius of 0.5. In other words, the sphere is the inscribed sphere of a unit cube that has been translated 0.5 in the z dimension.
Before collision checking, these spheres are scaled and translated to match the agent’s scale and translation.

These spheres do not accurately approximate many of the agents. Many man-made objects have rectangular edges, such as the fortress model. For such agents, a bounding box would provide a closer approximation to the actual vertices of the agent than a bounding sphere. The spheres also do not substantially cover agents that extend significantly outside of the unit cube. For example, after the initial scaling and centering of the fireman model’s vertices, these vertices extend out to more than twice the height of the unit cube. Thus the sphere used to do collision checking for the fireman only detects collisions that occur around his feet and lower body. Some of the drawbacks of the current collision detection system are shown in Figure 2-3 below.
Figure 2-3: Fireman and dragon model compared to current collision sphere. (Top left) Top view of fireman model compared to the sphere. (Bottom left) Side view of fireman model compared to the sphere. The fireman is much taller than the sphere. (Top right) Top view of dragon model compared to the sphere. (Bottom right) Side view of dragon model compared to sphere. The dragon is much smaller than the sphere.
3 TightestFitCollider

This section describes the first new collision detection system, beginning with the motivation (3.1) and an overview of the design (3.2). Next the details of the implementation are discussed, including the new data structures created (3.3), the algorithms for finding tight-fitting bounding structures (3.4), transforming these bounding structures (3.5), and intersection tests between the bounding structures (3.6). Finally, this section concludes with a description of how the TightestFitCollider is incorporated into the existing framework (3.7) and some notes on its configurable parameters (3.8).

3.1 Motivation

The motivation for the TightestFitCollider is that the spheres used in the current collision detection system are not good approximations for most of the models in StarLogo Nova. Some of the models, such as the fireman, extend outside the sphere, and for many more models, such as the dragon, the sphere contains a lot of additional empty space. The goal of the TightestFitCollider is to use a small set of basic bounding structures that can approximate a larger variety of models more accurately. These bounding structures need to be basic geometric shapes, so that the pairwise intersection tests between them are still fast. The potential bounding structures explored are spheres, boxes, cylinders, and capsules.
3.2 Design overview

There are two stages of computations for the TightestFitCollider. The first stage is the loading stage. As a game loads in StarLogo Nova, the tightest fitting bounding structure is computed and stored for each of the models used in that particular game. Finding the best bounding structure is pre-computed because it takes too long to do during gameplay. It is also pre-computed and stored because it is always the same for each instance of a model. For example, even if a game contains fifty dragons, the bounding structure for the dragon model only needs to be computed once.

The second stage is the gameplay stage. As a game is being played, the TightestFitCollider must determine if pairs of agents are colliding. For each pair of agents, the collider first accesses the bounding structure of each agent's model. Next each bounding structure is transformed (scaled, rotated, and translated) to match the current size, heading, and location of that agent. Finally, the transformed bounding structures are passed to a pairwise intersection test that determines if they intersect. If the bounding structures intersect, the TightestFitCollider returns that the two agents are colliding.

3.3 Data structures

The TightestFitCollider led to the creation of the following new data structures:

1. BoundingStructuresLibrary

This class contains dictionaries that map models (using their shape names) to bounding structures.
2. **BoundingStructure**

Any basic geometric shape that is used as a bounding structure implements this interface. This interface requires each implementing class to contain a method for finding its own volume, as well as methods for scaling, rotating, and translating.

3. **Sphere**

This class implements **BoundingStructure**. It stores the center of the sphere as a `Vector3D` and the radius as a `Number`.

4. **Box**

This class implements **BoundingStructure**. It stores data for an arbitrarily oriented rectangular prism. The center of the box is stored as a `Vector3D`. The three local axes of the box are also stored as `Vector3Ds`. The size of the box is stored as three half-extents along the three local axes. The half-extents are `Numbers`.

5. **Capsule**

This class implements **BoundingStructure**. It stores data for an arbitrarily oriented capsule. A capsule can be thought of as a sphere that has been swept along a line segment or as a cylinder capped with hemispheres. A capsule has the property that any point on its surface is a fixed distance from its line segment. A capsule is stored as the two endpoints of its line segment and a radius. The endpoints are stored as `Vector3Ds` and the radius is a `Number`.

**Note:** A cylinder class was originally implemented as well. However, due to difficulties with cylinders, which are discussed more in section 3.6.1, cylinders are not used in the final implementation of TightestFitCollider.
3.4 Finding tight fitting bounding structures

This section describes various algorithms for finding tight fitting bounding structures, which include bounding spheres, bounding boxes, bounding cylinders, and bounding capsules.

3.4.1 Bounding spheres

The current collision detection system uses spheres, but the spheres are not always bounding spheres because they do not always complete enclose the model. For example, as discussed earlier, the fireman model extends outside of the sphere. This section takes a look at two algorithms that may provide better fitting spheres by fully enclosing the models. Each model contains a set of vertices, and a bounding structure bounds the model if and only if all of the vertices fall within that bounding structure.

The first algorithm is the naïve method for finding a bounding sphere. In a first pass through the vertices, the minimum and maximum values along each of the three dimensions is stored. The center of the bounding sphere is calculated as the average of the minimum and maximum values in each dimension. Next, calculating the radius of the bounding sphere takes a second pass through the vertices. The second pass finds the maximum distance between the center and any vertex. The maximum distance becomes the radius. This bounding sphere is likely larger than the exact minimal bounding sphere, but it provides a reasonable approximation and encloses all the vertices. The volumes of the bounding spheres for various models are given in section 6, specifically in Figure 6-1.
The second algorithm is the Ritter algorithm for finding bounding spheres [7]. This algorithm is similar to the naïve one in that it requires two passes through the vertex set. However, the details are slightly more complicated. An initial sphere is calculated by choosing two vertices that are far from each other and using these two vertices to define the diameter. There are a few ways that the two vertices can be chosen. One strategy is to select any random vertex to be the first vertex, and then go through the other vertices to find the vertex that is farthest away from the vertex. The vertex that is farthest away becomes the second vertex. Another strategy is to find the minimum and maximum values in each dimension, similar to the naïve algorithm. Then for whichever dimension has the greatest range, the two vertices corresponding to the minimum and maximum values are chosen. After the initial sphere has been calculated through one of these strategies, a second pass through the vertices is required. In the second pass, each vertex is checked to see if it lies within the sphere. If it lies outside the sphere, both the center and the radius of the sphere are updated, so that the new sphere is just large enough to cover both the old sphere and the current vertex. Thus another way that this algorithm differs from the naïve one is that both the center and the radius are modified throughout the second pass through the vertices. In practice, for the subset of models from StarLogo Nova that were sampled, the naïve algorithm found tighter-fitting spheres compared to the Ritter algorithm on average. The volumes of the bounding spheres calculated by the Ritter algorithm are given in section 6, in Figure 6-1.
3.4.2 Bounding boxes

This section discusses algorithms for finding three types of bounding boxes: axis-aligned bounding boxes, z-aligned bounding boxes, and oriented bounding boxes. An axis-aligned box is a box whose local axes lie parallel to the axes of the world coordinate system. Finding an axis-aligned box for a set of vertices is straightforward and fast. It requires one pass through the vertices, keeping track of the minimum and maximum values along each of the three dimensions. These six values define the axis-aligned box.

A z-aligned box is a box that has one local axis aligned with the z-axis of the world coordinate system. The other two local axes of the box can lie anywhere in the xy-plane, as long as they are perpendicular to each other. Z-aligned boxes are useful for fitting models that may rotate only around the z-axis. Currently, all agents in StarLogo Nova fit this criterion, but StarLogo Nova plans to allow rotations in any direction in the future. Thus z-aligned boxes are useful for StarLogo Nova currently, and may continue to be useful if there is subset of models that only rotate in the xy-plane.

The algorithm for finding a z-aligned bounding box is a brute-force algorithm. First, the possible rotations in the xy-plane are discretized. For example, the algorithm may look at rotations every five degrees or rotations every ten degrees. For each possible rotation, the first step is to rotate the vertices by that rotation. Then an axis-aligned box is calculated for the rotated vertices. Finally, the axis-aligned box is rotated in reverse, so that it fits as tightly around the original vertices. As the algorithm proceeds through the possible rotations, it keeps track of
the smallest (by volume) z-aligned bounding box so far. At the end, it outputs the
box that has the smallest volume because that means it's the best (tightest) fit.

The third type of bounding box is an oriented bounding box. An oriented
bounding box is a box that is oriented along the direction of maximum variation in
the vertex set. Typically, an oriented bounding box provides a much tighter fit
compared to an axis-aligned box.

The first algorithm that was considered for finding oriented bounding boxes
uses covariance matrices. These matrices are useful because their eigenvectors
describe the primary and secondary axes of maximum variance in the vertex set [6].
Eigenvectors are also perpendicular to each other. Thus the eigenvectors of the
covariance matrix can be used as the local axes of the oriented bounding box.

Although this algorithm seemed promising, there are several drawbacks to
this approach. One challenge is specific to Actionscript3, and thus makes the
covariance matrix method impractical for use in StarLogo Nova. In particular, the
first step of finding the covariance matrix is quite straightforward. However, after
the matrix is found, its eigenvectors must be extracted. Actionscript3 does not have
efficient built-in capabilities for finding the eigenvalues or eigenvectors of a matrix.
Implementing a function to find the eigenvectors usually makes use of a numerical
solver. Unfortunately, Actionscript3 does not have a numerical solver either, and
implementing one would be outside the scope of this thesis. Furthermore, this
language-specific issue is not the only drawback to using the covariance matrix
algorithm.
A second drawback to using the covariance matrix algorithm is that it is highly sensitive to the distribution of the vertices. In particular, the internal vertices of the model influence the covariance matrix even though the bounding box should only be dependent on the external vertices of the model (the vertices on the convex hull). In practice, it has been found that the covariance matrix method often does not lead to tightly fitting oriented bounding boxes [3].

The alternate algorithm for finding a tight-fitting oriented bounding box is a brute-force algorithm similar to the one described above for finding a z-aligned bounding box. As before, the space of possible rotations is first discretized. Then for each possible rotation, the vertices are rotated, an axis-aligned bounding box is calculated, and then this axis-aligned bounding box is rotated in reverse, so that it fits the original non-rotated vertices. The oriented box with the smallest volume from all of the rotations is chosen as the tightest fitting oriented bounding box. Compared to the algorithm for finding a z-aligned bounding box, this algorithm is much slower since the space of potential rotations is much larger. The z-aligned bounding boxes only examined rotations around the z-axis, but the oriented bounding boxes must take into account any combination of rotations around the x-axis, y-axis, and z-axis. The runtimes of this algorithm for various models are discussed in section 6.

### 3.4.3 Bounding cylinders

The motivation for considering bounding cylinders is that they can be a better fit than boxes or spheres for models of people. Finding a bounding cylinder is
straightforward given a bounding box. The longest axis of the bounding box can be used to find the long axis and height of the cylinder. In particular, the long axis of the cylinder is positioned at the center of the bounding box and aligned along the longest axis of the bounding box. The height of the cylinder is the length of the longest side of the bounding box. Then finding the radius of the cylinder requires one additional pass through the vertices. In particular, the radius is the maximum distance from the long axis of the cylinder to any vertex. Although finding bounding cylinders is easy, bounding cylinders are not used in the TightestFitCollider because the intersection tests with rotated cylinders are computationally expensive and thus impractical. This limitation is discussed in more detail in section 3.6.1.

### 3.4.4 Bounding capsules

A bounding capsule is similar to a bounding cylinder, except that it has two hemispheres capped on the ends of the cylinder. Thus one way to find a bounding capsule is to first find the bounding cylinder and then extend the cylinder at the ends using the cylinder’s radius. Although these steps would find a bounding capsule, they would not necessarily find a tight fitting bounding capsule. Since the bounding cylinder already fully encloses the vertices, the hemispheres of a bounding capsule constructed this way would be completely empty. Figure 3-1 illustrates this issue.

An alternative algorithm for finding a tight fitting bounding capsule begins with a bounding box for the vertices. Any type of bounding box can be used. An axis-aligned bounding box will lead to an axis-aligned bounding capsule, whereas an
oriented bounding box will lead to an oriented bounding capsule. Similar to the cylinder construction, the long axis of the capsule is positioned at the center of the bounding box and aligned with the longest axis of the bounding box. Then the radius is computed as the largest distance from the long axis of the capsule to any vertex. This algorithm differs, however, in the calculation of the endpoints of the capsule line segment. Instead of setting the endpoints to lie on the surface of the bounding box (such that the length of the capsule line segment is equal to the length of the longest side of the bounding box), the capsule is first inscribed inside the bounding box, as shown below in Figure 3-1. This capsule is the smallest a bounding capsule could be. Then this initial bounding capsule is slowly grown as necessary, similar to the way the Ritter sphere from section 3.4.1 is grown. In particular, each vertex is checked to see whether it lies within the capsule or not. If it does not, the endpoints of the capsule are expanded just enough to cover that vertex. This update procedure requires one additional pass over the vertices, starting from a bounding box.
Figure 3-1: The fireman model with its bounding box (blue) and various bounding capsules (red). (Left) A bounding capsule whose hemispheres extend out from the bounding box. The hemispheres are empty and do not contain any of the model's vertices. (Center) A capsule inscribed inside the bounding box. It does not necessarily enclose all the vertices. (Right) A tight-fitting bounding capsule that results from updating the capsule in the center.

3.5 Transforming bounding structures

For a given agent-agent collision check, the bounding structure for each agent must be scaled, rotated, and translated to match the size, heading, and current position of the agent. The original agent model is not necessarily centered at the world origin, since models are usually positioned so that their z values are non-negative. Thus the bounding structure calculated is likewise not centered at the world origin.

Since the center of a bounding structure may not be at the world origin, there is a difference between scaling the structure relative to the world origin versus scaling the structure relative to its own center. Similarly, rotating the bounding
structure around the origin is different from rotating the structure around its
center. Currently, agents are rotated around their models’ origins. For most models,
the model’s origin is located at its center in the x and y dimensions and at its
minimum in the z dimension. The transformations applied to the bounding
structures need to match the transformations applied to the agents.

In the future, when rotations in any direction are allowed, it may be more
natural to rotate agents around their own centers. If changes are made to the way
agents are transformed, the transformations to the bounding structures must
change as well, so that the bounding structures continue to fit the agents
appropriately.

3.6 Intersection tests

This section describes the intersection tests between each possible pair of the
bounding structures described in the previous section. These intersection tests need
to be very fast, since they run in real-time as the game is being played.

3.6.1 Feasibility of cylinder intersection tests

Although cylinders are a good fit for human characters in games, their intersection
tests are only efficient when they are aligned along one of the major world axes. For
arbitrarily oriented cylinders, finding intersections requires a numerical solver [2].
There are no built-in numerical solving capabilities in Actionscript3, and they would
likely be too slow for real-time collision detection anyway. Thus agents that are
modeled as cylinders would not be able to be rotated arbitrarily in any direction.
StarLogo Nova is moving towards allowing any agent to be rotated in any direction
to allow users to build games that include fluid movements, such as flying. A close approximation to an oriented cylinder is an oriented capsule. Thus the current implementation of the TightestFitCollider focuses only on intersection tests between spheres, boxes, and capsules.

### 3.6.2 Sphere-Sphere

The sphere-sphere intersection test is very straightforward and fast. The distance between the centers of the spheres is compared to the sum of the radii. If the distance is less than or equal to the sum of the radii, the spheres intersect. Taking a square root is one of the slower basic math operations, so this can be avoided by comparing the square of the distance between the sphere centers to the square of the sum of the radii. The sphere-sphere intersection test is illustrated in Figure 3-2 below.

![Figure 3-2: Spheres with the distance between their centers shown in red. (Left) The spheres intersect because the distance between their centers is less than the sum of their radii. (Right) The spheres don’t intersect because the distance between their centers is greater than the sum of their radii.](image)

Figure 3-2: Spheres with the distance between their centers shown in red. (Left) The spheres intersect because the distance between their centers is less than the sum of their radii. (Right) The spheres don’t intersect because the distance between their centers is greater than the sum of their radii.
3.6.3 Capsule-Capsule

The capsule-capsule intersection test is based on a similar idea to the sphere-sphere intersection test, but it is more complicated to implement. The basic idea is to find the shortest distance between the inner line segments of the two capsules. Then this shortest distance can be compared to the sum of the radii of the capsules to determine if the capsules intersect. To find the shortest distance between the two line segments, the shortest distance is found between the two lines containing the line segments. Then if the closest point on either of the lines lies outside the line segment, the endpoint of the line segment becomes the new closest point.

The first step in finding the shortest distance between two lines is to parameterize them. Suppose we have two capsules, $a$ and $b$, and each capsule has the fields $point_1$ and $point_2$ that define its line segment. Then the capsule lines can be parameterized as follows:

$$l_1 = a.\text{point}_1 + t_1v_1, \text{ where } v_1 = a.\text{point}_2 - a.\text{point}_1$$

$$l_2 = b.\text{point}_1 + t_2v_2, \text{ where } v_2 = b.\text{point}_2 - b.\text{point}_1$$

Now suppose the shortest distance between the two lines occurs at $t_1 = a_{\text{closest}}$ and $t_2 = b_{\text{closest}}$. We are interested in finding $a_{\text{closest}}$ and $b_{\text{closest}}$ in order to find the shortest distance between the two lines.

If the two lines are skew, then the line defined by the points $a_{\text{closest}}$ and $b_{\text{closest}}$ is perpendicular to both $l_1$ and $l_2$. This property can be used to solve for $a_{\text{closest}}$ and $b_{\text{closest}}$. Let $l_3$ be the line defined by $a_{\text{closest}}$ and $b_{\text{closest}}$. Then $l_3$ can be written as follows:

$$l_3 = (b.\text{point}_1 + b_{\text{closest}}v_2) + t_3v_3, \text{ where}$$
\[ v_3 = (a.\ point_1 + a_{closest} v_1) - (b.\ point_1 + b_{closest} v_2) \]

Since \( l_1 \perp l_3 \) and \( l_2 \perp l_3 \), it follows that \( v_1 \cdot v_3 = 0 \) and \( v_2 \cdot v_3 = 0 \). Substituting for \( v_3 \) leads to the following two equations:

\[ v_1 \cdot (a.\ point_1 + a_{closest} v_1 - b.\ point_1 - b_{closest} v_2) = 0 \]
\[ v_2 \cdot (a.\ point_1 + a_{closest} v_1 - b.\ point_1 - b_{closest} v_2) = 0 \]

Rearranging these two equations leads to:

\[ (v_1 \cdot v_1) a_{closest} - (v_1 \cdot v_2) b_{closest} = -(v_1 \cdot (a.\ point_1 - b.\ point_1)) \]
\[ (v_1 \cdot v_2) a_{closest} - (v_2 \cdot v_2) b_{closest} = -(v_2 \cdot (a.\ point_1 - b.\ point_1)) \]

Now there is a system of two equations, which can be solved to give the two unknowns, \( a_{closest} \) and \( b_{closest} \). To simplify the expressions, the following constants are used:

\[ c_1 = v_1 \cdot v_1 \]
\[ c_2 = v_1 \cdot v_2 \]
\[ c_3 = v_2 \cdot v_2 \]
\[ c_4 = v_1 \cdot (a.\ point_1 - b.\ point_1) \]
\[ c_5 = v_2 \cdot (a.\ point_1 - b.\ point_1) \]

Now the system of two equations can be written as:

\[ c_1 a_{closest} - c_2 b_{closest} = -c_4 \]
\[ c_2 a_{closest} - c_3 b_{closest} = -c_5 \]

Solving this system yields \( a_{closest} = \frac{c_2 c_5 - c_3 c_4}{c_1 c_3 - c_2^2} \) and \( b_{closest} = \frac{c_1 c_5 - c_2 c_4}{c_1 c_3 - c_2^2} \). These are the values of the parameters \( t_1 \) and \( t_2 \), which can be inserted back into the parameterized equations of the two lines to find the closest point on each line to the
other. As mentioned earlier, these points also need to be checked to see if they lie outside the line segment. In order to lie within the line segment, \(0 \leq a_{closest} \leq 1\) and \(0 \leq b_{closest} \leq 1\). Thus clamping the values to this range leads to the closest points on the line segment.

The solution described above applies to two lines in 3D that are skew. The lines could alternately be intersecting or parallel. The case when the lines are intersecting is a special case of when the lines are skew. The values for \(a_{closest}\) and \(b_{closest}\) will lead to points on each line that are actually the same point. Thus the only problematic case is when the lines are parallel. This case can be distinguished by examining the final equations for \(a_{closest}\) and \(b_{closest}\). These equations each contain a fraction. When the lines are parallel, the value of the denominator of these equations will be 0. Thus checking this denominator before dividing by it will catch the case when the lines are parallel.

If the denominator is equal to 0 and the two lines are parallel, then the distance between them is fixed. The first step is to choose any point on the first line. Projecting that point to the second line provides the closest point on the second line. For example, \(a_{closest}\) can be set to equal 0. Then using the values defined earlier, the projection of \(a_{closest}\) onto \(l_2\) can be written as \(b_{closest} = \frac{c_5}{c_3}\).

In the final step, the shortest distance between the two lines is compared to the sum of the capsule radii. Once again, the square of the distance and the square of the sum of the radii are used to avoid taking square roots.
3.6.4 Box-Box

The box-box intersection test makes use of the Separating Axis Theorem [3]. In 3D, if two boxes are not colliding, then there is a plane that separates them. The separating axis lies along any normal of this plane. If the two boxes are projected to this axis, then their projected intervals do not overlap if and only if the corresponding plane separates them. Thus to determine if a certain plane separates two boxes, we only need to find a corresponding separating axis, project the two boxes to this axis, and check if the projected values overlap. Figure 3-3 illustrates an example of a separating axis and the projected intervals.

Figure 3-3: Two oriented boxes projected to a separating axis. The blue dotted lines show the minimum and maximum values of the projections of the boxes onto the separating axis. The projected intervals are shown in red. The two boxes can be separated by a plane perpendicular to the separating axis.

At first glance, it may seem that there are six possible separating axes. The first box could lie completely to the left or completely to the right of the second box. In that case, the local x-axis of the second box is the separating axis. Alternately, the
first box could lie completely above or completely below the second box. Then the local y-axis of the second box is the separating axis. Finally, the first box could lie completely in front of or completely behind the second box. If the two boxes were both axis-aligned boxes, it would be sufficient to check these three axes. The local axes of both boxes would be the same as the world axes. Thus checking the local x-axis of the first box would be the same as checking the local x-axis of the second box. When both boxes are arbitrarily oriented, however, we also need to check the local axes of the other box. Thus the three local axes of each box form the first six possible axes of separation.

These six axes do find non-collisions between two boxes in most cases. However, there are special cases when two boxes are almost colliding edge-to-edge. In these cases, their projected intervals overlap for all of the six axes discussed above. The actual axis of separation is given by the cross product of one of the local axes of the first box and one of the local axes of the second box. An example of this case is shown in Figure 3-4. Thus to find these non-collisions, we must add nine more potential axes of separation. These nine are the cross products between the three local axes of the first box and the three local axes of the second box.

Altogether there are fifteen possible axes of separation between two arbitrarily oriented boxes. For each axis, we project the eight corner points of each box onto the axis. If the projected intervals don’t overlap, then the axis is a separating axis for the two boxes. At this point it is not necessary to continue checking the other axes of separation because we know that the two boxes do not collide. However, if the projected intervals do overlap, then we must continue with
the other potential axes of separation. If the two boxes overlap for all fifteen separating axes, then the two boxes are colliding.

Thus this intersection test exits quickly for two boxes that are not colliding. It is slowest when two boxes are colliding because then it must run through all fifteen cases. One potential optimization is to only use the first six axes discussed. According to one study, only 15% of box-box intersections enter the latter nine cases [3]. Of these, if about half are colliding and about half are not colliding, then using only the first six cases will result in a false positive 7.5% of the time. In other words, 7.5% of the time, a collision will be detected even though the boxes do not actually collide. Using this optimization would make this intersection test over twice as fast.

Figure 3-4: Two boxes that are almost colliding edge-to-edge. (Left) There is a small gap between their closest edges. (Right) A view of the same boxes from the top. The black arrow shows the cross product of the two local axes (one from each box) that is the separating axis. Image modified from [4].
3.6.5 Sphere-Capsule

The sphere-capsule intersection test is similar to the sphere-sphere and capsule-capsule intersection tests. The basic idea is to find the shortest distance between the center of the sphere and the line segment of the capsule. Then this distance can be compared to the sum of the radii of the sphere and capsule to see if they are colliding. As before, the square of the distance and the square of the sum of the radii are used to avoid taking square roots.

Finding the shortest distance between the center of the sphere and the line segment of the capsule is also straightforward. The closest point on the line segment is simply the projection of the sphere center onto the line containing the line segment. If this closest point lies outside the line segment, as in Figure 3-5 below, then the closest endpoint of the line segment is used instead.

![Figure 3-5: Sphere-Capsule intersection tests. (Left) A sphere and capsule intersect. The capsule line segment is shown in blue, and the line containing it shown in dotted blue. The projection of the sphere center (dotted red) lies outside the line segment. The shortest](image)
distance from sphere center to capsule line segment is shown in red. (Right) The red line shows the shortest distance from sphere center to capsule line segment.

3.6.6 Sphere-Box

The sphere-box intersection test is an implementation of Arvo's algorithm [1]. This algorithm begins by finding the shortest distance between the box and the center of the sphere. First, the sphere is transformed into the local coordinate system of the box, so that the box can be treated as an axis-aligned box. Then each dimension is examined in turn. For each dimension, if the sphere center lies outside the range of the box (either less than the box minimum or greater than the box maximum), the distance between the sphere center and the box extremum is added to the total distance. If the sphere center lies within the range of the box in a particular dimension, then that dimension does not contribute any distance to the total shortest distance between the sphere center and the box.

After the shortest distance between the box and the center of the sphere has been found, this distance is compared to the radius of the sphere. If the distance is less than or equal to the radius of the sphere, the box and the sphere are intersecting. As before, the square of the distance is compared to the square of the sphere radius to avoid taking unnecessary square roots.
Figure 3-6: Sphere-Box intersection tests. (Left) The sphere center lies outside the range of 
the box along its local x-axis. The red line shows the shortest distance from the sphere center 
to the box along this axis. (Right) The sphere center lies outside the range of the box along 
both its local axes. The red lines show the shortest distances in each dimension.

3.6.7 Capsule-Box

The capsule-box intersection test is similar to the box-box intersection test. As 
before, we would like to find a separating axis that corresponds to a plane that 
separates the capsule and the box, if one exists. There are six potential axes of 
separation. The first three are the three local axes of the box. These determine 
whether the capsule lies completely to the left, right, above, below, in front of, or 
behind the box. The second three are the cross products of the three local axes each 
with the long axis of the capsule. These address the cases when the capsule and the 
box are almost colliding edge-to-edge.

There is also one slight adjustment to the process described in the box-box 
intersection test, since here the other object is a capsule and not another box. After
the capsule’s line segment is projected to the axis of separation, we extend the
capsule’s interval on either end by its radius. A capsule extends a radius-distance
out in every direction from the line segment. Therefore regardless of how the
capsule’s line segment aligns with the axis of separation, and whether its projection
is of length zero, the entire length of the line segment, or anywhere in between, the
actual capsule will extend out a distance equal to the radius in both directions, as
shown in Figure 3-7.

![Figure 3-7: A capsule projected to a separating axis. The blue solid line is the capsule line segment. The blue dotted lines show its projection. The green solid lines are radii of the capsule. The green dotted lines show their projections. The red solid line shows the overall projection of the capsule.](image)

### 3.7 Integration into existing framework

The current collision system is called the BinCollider. In the BinCollider, the world is
split into regions or “bins” as a first pass of filtering possible collisions, so that only
agents near each other (in the same bin) require the more complicated math of
collision checking. The TightestFitCollider is built on top of BinCollider in order to
take advantage of the binning system. Integrating into the existing framework
requires only a small number of changes. First, the Collision class stores the collision
system used as one of its fields. The TightestFitCollider implements the
CollisionSystem interface that previously existed. Thus, the collision system field
can be set to TightestFitCollider instead of BinCollider.

Each type of model must also find its tightest fitting bounding structure after
its vertices are loaded. In the BoundingStructuresLibrary, there is a method that
calculates and stores the tightest fitting bounding structure. This method takes in a
model name and a set of vertices. Currently, it compares the volumes of the naïve
sphere, oriented box, and oriented capsule. However, the specific algorithms it
compares between can be adjusted at any time. For example, if rotations are only
allowed around the z-axis, then the z-aligned box can be used instead of the oriented
box. This method must be called from each type of model while it loads: built-in
models, ObjModels, and Collada models.

Finally, the GreatLand class stores an instance of the
BoundingStructuresLibrary, as well as a few different flags and variables that allow
the TightestFitCollider to be flexible and configurable. These are discussed more in
the next section.

3.8 Configurability

The following flags and variables set in GreatLand allow the TightestFitCollider to
be configurable:
1. **loadTightestFit** is a flag that determines whether the various models should call the method to calculate and store the tightest fit bounding structure. This should be set to false if a different collision system is being used.

2. **loadAllBoundingStructures** is a flag that determines whether all of the bounding structures calculated should be stored (in different dictionaries) instead of just the tightest fit bounding structure. This may useful in future work to do checks between bounding spheres before moving on to more expensive checks between bounding boxes.

3. **orientedBoxDegIncrement** is a variable that determines the step size for discretizing the space of all possible rotations when finding oriented boxes. This is set to ten degrees currently to provide a reasonable performance rate.
4 HierarchicalCollider

This section describes the second new collision detection system, beginning with the motivation (4.1) and an overview of the design (4.2). Next the details of the implementation are discussed, including the new data structures created (4.3), the hierarchy construction (4.4), hierarchy transformation (4.5), and hierarchy traversal (4.6). Finally, this section concludes with a description of how the HierarchicalCollider is incorporated into the existing framework (4.7) and some notes on its configurable parameters (4.8).

4.1 Motivation

The motivation for the HierarchicalCollider is that the TightestFitCollider may still not provide as precise collision detection as desired. For example, consider the dragon model shown in Figure 4-1. The tightest fitting bounding structure, which is a bounding box, still includes a lot of extra space. This may be especially frustrating if this model is the main character of a game. Then the user may be particularly sensitive to incorrect collision detection, since most of the attention is on the main character of the game. One way to improve the precision of collision detection is to use a hierarchical collision system instead. The cost of this improved precision is that hierarchical systems are much slower. The tradeoffs and performance metrics are described in sections 5 and 6 respectively.
Figure 4-1: Dragon model with its tightest fitting bounding box (blue). There is a lot of extra space in the bounding box due to the shape of the dragon.

4.2 Design overview

Similar to the TightestFitCollider, there are two stages of computations for the HierarchicalCollider. As before, the first stage is the loading stage. As a game loads in StarLogo Nova, the bounding structure hierarchy is computed and stored for each of the models used in the particular game. Finding these hierarchies would take too long to do during gameplay. The hierarchy for a particular model is also the same for each agent that uses that model, and thus it does not need to be calculated repeatedly. The current implementation of the HierarchicalCollider creates a hierarchy of bounding boxes.

The second stage is the gameplay stage. As a game is being played, the HierarchicalCollider must determine if pairs of agents are colliding. For each pair of agents, the collider first accesses the hierarchy of each agent’s model. Next both
hierarchies are traversed. For each pair of nodes that is compared, the bounding boxes stored at those nodes are accessed. Then these bounding boxes are transformed (scaled, rotated, and translated) to match the current size, heading, and location of the agent. Finally the transformed bounding boxes are passed to the appropriate intersection test. If the bounding boxes do not intersect, then there is no need to go to the next level of the hierarchy for each of those nodes. However, if they do intersect, the same steps repeat for the children of these nodes. The following sections describe the details of hierarchy construction, hierarchy transformation, and hierarchy traversal.

4.3 Data structures

The HierarchicalCollider required the addition of only two new data structures.
Many of the data structures from the TightestFitCollider were re-used.

1. HierarchiesLibrary
This class contains a dictionary that maps models (using their shape names) to the root nodes of the bounding box hierarchies.

2. HierarchyNode
This class stores all the data for a node in the bounding box hierarchy. It stores a bounding box for the vertices at this node and the number of vertices that are enclosed by that bounding box. It also stores its left and right children, which are also HierarchyNodes. These are stored as null if the node is a leaf. Finally, it stores a Boolean isLeaf.
4.4 Hierarchy construction

In the loading stage, the hierarchical collision system constructs a hierarchy of bounding boxes for each model. Proceeding down the hierarchy, each level contains more bounding boxes that are each smaller. The idea is that each level of the hierarchy removes some of the extra space that was present in the preceding level of the hierarchy. Thus the bounding boxes at lower levels provide a closer approximation to the shape of the model compared to the bounding structures at higher levels.

There are a couple design questions involved in constructing a hierarchy. The first question is what types of bounding structures the hierarchy should store. As discussed already, the current implementation constructs a hierarchy of bounding boxes. Bounding boxes are often chosen because they are the tightest fitting shapes for most models. Thus a hierarchy made of bounding boxes is likely to be more precise at an earlier level compared to a hierarchy of bounding spheres. Then the hierarchy does not need to be traversed as deeply to reach the same level of precision.

A second question is how the vertices at a node should be split into the vertices of the two child nodes. The goal is that the total volume of the child nodes’ bounding boxes should be less than the volume of the parent node’s bounding box. Many strategies choose to split along the longest axis of the bounding box, since this is the direction of greatest variation. A few strategies try splitting along each of the three axes of the bounding box, and choose the axis that leads to bounding boxes of
the smallest total volume in the next level. However, these latter strategies slow down the hierarchy construction by a factor of three.

The axis to split is not the only question involved in creating the child nodes. There are also various splitting points along the splitting axis. One strategy is to split by the median, so that an equal number of vertices end up in each box of the second level. Another strategy is to split by the mean of the vertices. Although using the median leads to more balanced trees, using the mean leads to trees that are faster to traverse on average [3]. Thus one of the strategies I implemented splits each bounding box by the mean along the longest axis. Finally, a third strategy that is much faster than either of the other two mentioned so far is to split at the halfway point of the longest axis. This strategy leads to much faster hierarchy construction, since it does not require an additional pass through the vertices for each node of the hierarchy. This strategy has also been implemented in the HierarchicalCollider. Thus either splitting by mean or splitting by halfway point can be used to construct the hierarchies. Since the hierarchies are constructed during loading time, which is still time that the user must wait, the HierarchicalCollider currently uses the splitting by halfway point method. However, the system is configurable and can be switched to using means.

Putting these pieces together, the hierarchy is constructed as follows. First, a bounding box is computed for the model overall. This first bounding box is stored at the root node of the hierarchy. Then the halfway point of the longest axis of the bounding box is determined. Next the vertices are partitioned into two sets, depending on if they are less than or greater than the halfway point. Finally, two
more bounding boxes are constructed, one for each vertex set from the partitioning. These bounding boxes are stored as the children of the root node. This process repeats at each node, and the implementation is recursive.

There are two terminating cases for the recursion. One configurable parameter is the maximum depth of the hierarchy tree. Once this maximum depth is reached, a node becomes a leaf node for the hierarchy. Another configurable parameter is the minimum number of vertices that can be split further. Once the number of vertices is less than or equal to this minimum, the node is not split any further.

![Figure 4-2: The first three levels of the dragon model hierarchy. Each level is split halfway along the longest axis.](image)

**4.5 Hierarchy transformation**

Since the hierarchies for each model are constructed during the loading stage of the program, the hierarchies fit the model when it is centered at the origin and has not been scaled or rotated. The actual agents in the game move around and could have been scaled or rotated. Thus the transformations applied to the agent must be
applied to its hierarchy as well, so that the bounding structures continue to fit the agent.

Each bounding box already contains methods that scale, rotate, and translate it. These methods are called to transform the boxes in the hierarchy. Rather than transforming the entire hierarchy upfront at the beginning of collision detection, each box is transformed just before its intersection with another box is checked. Thus extra time is not wasted transforming boxes at the lower levels of the hierarchy that may never be reached in a given traversal.

4.6 Hierarchy traversal

The basic idea of traversal is that beginning at the two root nodes of the two hierarchies, the bounding boxes of those nodes are transformed and then checked for intersection. If those bounding boxes intersect, then the same process repeats for each pair of children nodes (one child from each hierarchy). If the bounding boxes don’t intersect, then the method returns from that part of the tree. The traversal also checks to see if the nodes are leaf nodes before continuing on. If there is an intersection between two leaf nodes, then the two agents are colliding.

One design question for traversal is whether to follow a breadth-first traversal approach or a depth-first traversal approach. According to related research, depth-first traversal approaches tend to traverse the tree faster in the average case [3]. Since this traversal is happening in real-time as the game is being played, speed is of high importance. Thus the implementation of HierarchicalCollider also uses a depth-first approach. Another performance
optimization for hierarchy traversal is to specify a maximum depth for traversal. This max depth may be set globally for the system or on a per-agent basis. For the latter, it may be possible that certain agents, such as the main characters of games, require a higher level of precision, while other agents require only a few levels of hierarchy traversal to determine collision.

**4.7 Integration into existing framework**

Integrating the HierarchicalCollider into the existing framework requires similar changes as integrating the TightestFitCollider. The HierarchicalCollider is also built on top of the BinCollider in order to use binning as a preparatory step. These changes are summarized below.

The HierarchicalCollider implements the CollisionSystem interface that previously existed. Thus, the collision system field in the Collision class can be set to HierarchicalCollider instead of the current BinCollider.

Each type of model must also find its bounding box hierarchy after its vertices are loaded. In the HierarchiesLibrary, there is a method that calculates the bounding box hierarchy and stores the root node of the hierarchy. This method takes in a model name and a set of vertices. This method must be called from each type of model while it loads: built-in models, ObjModels, and Collada models.

Finally, the GreatLand class stores an instance of the HierarchiesLibrary, as well as a few different flags and variables that allow the HierarchicalCollider to be configurable. These are discussed more in the next section.
4.8 Configurability

The following parameters stored in GreatLand are configurable:

1. **loadHierarchies** is a Boolean flag that tells the various model types whether they should call the method to calculate and store the bounding box hierarchy. This should be set to false if the HierarchicalCollider is not being used.

The following parameters in HierarchiesLibrary are configurable:

1. **minVerticesToSplit** is the smallest number of vertices per node that should be split further into child nodes. This is currently set to 5.

2. **maxDepth** is the maximum number of levels that should be created in constructing a hierarchy. This is currently set to 10.

The following parameters in HierarchicalCollider are configurable:

1. **maxDepth** is the maximum number of levels that should be traversed for an agent’s hierarchy. Eventually, this may be set on a per-agent basis using blocks. It is currently set to 5.
5 CombinationCollider

This section describes the motivation and details of combining the TightestFitCollider and HierarchicalCollider into one system.

5.1 Motivation

The CombinationCollider is a combination of the TightestFitCollider and the HierarchicalCollider. The guiding observation here is that users may want some models to have much more precise collision detection compared to other models in the same game. For example, many games are first-person action games. This means that the user controls one character, and this character moves around exploring and interacting with the world. In games like this, the user may want collisions with the main character to be very precise. On the other hand, collisions between other agents in the background may not need to be as precise, since most of the player's attention is focused on the main character. Thus the main character might use hierarchical bounding boxes, while the remaining agents use their tightest fit bounding structures.

5.2 Integration of collision systems

The integration of the two collision systems is straightforward. All of the flags in GreatLand should be set to true, so that both hierarchies and tightest fit bounding structures are loaded for the models in the game. Then users should be able to set a trait for the agent that determines whether it should have hierarchical collision detection or not. The collision check will first determine whether either or both
agents are hierarchical. If neither is hierarchical, the collision check proceeds as it did in the TightestFitCollider. If both are hierarchical, the collision check proceeds as it did in the HierarchicalCollider. If one of the agents is hierarchical and the other is not, the collision check proceeds in a similar manner to the HierarchicalCollider. Only the hierarchical agent’s hierarchy is traversed, and at each node, the bounding box is checked for intersection against the tightest fitting bounding structure of the other agent. All of these pairwise intersection tests already exist because of the TightestFitCollider. Thus combining the two systems is simple.
6 Evaluation

In this section, I compare the performance of the three collision systems I developed to the current collision system. From the start, we knew that the new collision systems would be slower than the current collision system since they are more accurate. Here I evaluate the improved accuracy as well as the decreased performance.

6.1 Setup

In order to compare the overall performance of the three new systems to the existing system, I used a “game” in StarLogo Nova that has a constant number of collisions. The idea for this setup is credited to Daniel Wendel. For this setup, all of the agents in the game are assigned to the same bin. The agents are assigned to the position (1,1) because that position only falls into one bin (other positions on the grid, such as the origin, fall into multiple bins). Then a collision block is added that changes the color of an agent if it collides with any other agent. Thus all agents care about their collisions with all other agents. Finally, a “Move” push-button moves all agents forward on position each time it is clicked (i.e. the first click moves all of them to the position (2,1)). This setup ensures that the collision checking proceeds through all of the checks in the existing binning system to the point where an intersection test is run on each pair of agents. In other words, if there are \( n \) agents in this game, each time “Move” is clicked measures the time it takes for \( n^2 \) collisions.

For the current system, the game was tested with 100 agents, which means there are 10,000 collisions being checked. The time taken for 10,000 collisions is 30
milliseconds, so each collision check takes approximately 0.0003 milliseconds.

These numbers are true regardless of the type of agent tested, since all agents are approximated as spheres. This result also lines up with the speed of a sphere-sphere intersection test, which is around 0.0003 milliseconds.

6.2 TightestFitCollider

6.2.1 Volumes of bounding structures

Twelve models were chosen as representative models. These twelve include built-in models, man-made models (e.g. fortress), people-like models (e.g. person, fireman), plant models (e.g. kelp, tree), and animal models (e.g. bear, dragon). The dragon model is also included because it has one of the highest vertex counts for its model. Thus it is close to a worst-case for runtime from the current models.

Figure 6-1 shows the volumes of the various bounding structures for each of these twelve models. For comparison, the spheres used in the current system have volumes equal to 0.524 and do not necessarily bound the models at all. Each of these models fits inside a unit cube in the x and y dimensions, but may extend outside of the cube in the z dimension.

<table>
<thead>
<tr>
<th></th>
<th>Naïve sphere volume</th>
<th>Ritter sphere volume</th>
<th>Axis-aligned box volume</th>
<th>Z-aligned box volume</th>
<th>Oriented box volume (every 10 degrees)</th>
<th>Oriented capsule volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>0.524</td>
<td>0.524</td>
<td>0.989</td>
<td>0.953</td>
<td>0.959</td>
<td>0.596</td>
</tr>
<tr>
<td>Cube</td>
<td>2.721</td>
<td>5.216</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>3.052</td>
</tr>
<tr>
<td>Cylinder</td>
<td>1.803</td>
<td>4.577</td>
<td>1.131</td>
<td>1.121</td>
<td>1.123</td>
<td>1.412</td>
</tr>
<tr>
<td>Model</td>
<td>Volume 1</td>
<td>Volume 2</td>
<td>Volume 3</td>
<td>Volume 4</td>
<td>Volume 5</td>
<td>Volume 6</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>Bear</td>
<td>6.441</td>
<td>6.403</td>
<td>1.842</td>
<td>1.701</td>
<td>1.657</td>
<td>1.843</td>
</tr>
<tr>
<td>Dragon</td>
<td>0.598</td>
<td>0.576</td>
<td>0.298</td>
<td><strong>0.236</strong></td>
<td><strong>0.238</strong></td>
<td>0.593</td>
</tr>
<tr>
<td>Seagull</td>
<td>1.061</td>
<td>1.053</td>
<td>0.333</td>
<td>0.333</td>
<td><strong>0.282</strong></td>
<td>0.899</td>
</tr>
<tr>
<td>Tree</td>
<td>7.025</td>
<td>6.977</td>
<td>2.253</td>
<td>2.132</td>
<td><strong>2.056</strong></td>
<td>2.686</td>
</tr>
<tr>
<td>Kelp</td>
<td>0.524</td>
<td>0.524</td>
<td><strong>0.011</strong></td>
<td><strong>0.011</strong></td>
<td><strong>0.011</strong></td>
<td>0.026</td>
</tr>
<tr>
<td>Fortress</td>
<td>1.768</td>
<td>1.769</td>
<td><strong>0.501</strong></td>
<td><strong>0.501</strong></td>
<td><strong>0.501</strong></td>
<td>1.716</td>
</tr>
<tr>
<td>Spaceship</td>
<td>0.527</td>
<td>0.524</td>
<td>0.173</td>
<td><strong>0.167</strong></td>
<td><strong>0.168</strong></td>
<td>0.636</td>
</tr>
<tr>
<td>Fireman</td>
<td>7.909</td>
<td>7.861</td>
<td>1.733</td>
<td><strong>1.630</strong></td>
<td>1.635</td>
<td>2.517</td>
</tr>
<tr>
<td>Person</td>
<td>0.674</td>
<td>0.807</td>
<td>0.259</td>
<td><strong>0.220</strong></td>
<td><strong>0.221</strong></td>
<td>0.682</td>
</tr>
</tbody>
</table>

Figure 6-1: Volumes of various bounding structures for 12 representative models. The bolded values are the bounding structures of smallest volume for that model.

### 6.2.2 Performance of finding bounding structures

The same twelve models were tested to find how long it takes to determine the tightest fitting bounding structure. When tested, the bounding structures calculated and compared were the naïve sphere, oriented box (discretized every ten degrees), and the oriented capsule. Most of the time taken is in finding the oriented box. The time it takes to find the tightest fitting bounding structure is proportional to the number of vertices in the model.

<table>
<thead>
<tr>
<th>Model</th>
<th>Number of vertices</th>
<th>Time to load tightest fitting bounding structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>223</td>
<td>223 ms</td>
</tr>
<tr>
<td>Model</td>
<td>Vertices</td>
<td>Time</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>Cube</td>
<td>24</td>
<td>27 ms</td>
</tr>
<tr>
<td>Cylinder</td>
<td>320</td>
<td>318 ms</td>
</tr>
<tr>
<td>Bear</td>
<td>547</td>
<td>555 ms</td>
</tr>
<tr>
<td>Dragon</td>
<td>4070</td>
<td>3000 ms</td>
</tr>
<tr>
<td>Seagull</td>
<td>1154</td>
<td>924 ms</td>
</tr>
<tr>
<td>Tree</td>
<td>532</td>
<td>416 ms</td>
</tr>
<tr>
<td>Kelp</td>
<td>68</td>
<td>67 ms</td>
</tr>
<tr>
<td>Fortress</td>
<td>469</td>
<td>398 ms</td>
</tr>
<tr>
<td>Spaceship1</td>
<td>303</td>
<td>240 ms</td>
</tr>
<tr>
<td>Fireman</td>
<td>1458</td>
<td>1061 ms</td>
</tr>
<tr>
<td>Person</td>
<td>770</td>
<td>578 ms</td>
</tr>
</tbody>
</table>

Figure 6-2: Twelve models, each shown with its number of vertices and time taken to load the tightest fitting bounding structure.

### 6.2.3 Performance of accessing and transforming bounding structures

In this section, the time taken to access the tightest fitting bounding structure and transform (scale, rotate, and translate) it was measured for each of the twelve models measured in the previous section. The results for five of them are shown. The results of the other seven are comparable and were omitted for brevity. Models whose tightest fit is a sphere have slightly faster times than models whose tightest fit is a box, since there are fewer data values stored that need to be transformed. None of the sampled models have a capsule as the tightest fit, but transforming a
capsule is expected to take slightly longer than transforming a sphere and less time than transforming a box (since a sphere stores one vector and one number, a capsule stores two vectors and one number, and a box stores four vectors and three numbers).

<table>
<thead>
<tr>
<th>Model</th>
<th>Raw results</th>
<th>Time for accessing and transforming bounding structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube (tightest fit is box)</td>
<td>3736 ms for 1 million runs</td>
<td>0.003736 ms</td>
</tr>
<tr>
<td>Sphere (tightest fit is sphere)</td>
<td>2339 ms for 1 million runs</td>
<td>0.002339 ms</td>
</tr>
<tr>
<td>Cylinder (tightest fit is box)</td>
<td>4173 ms for 1 million runs</td>
<td>0.004173 ms</td>
</tr>
<tr>
<td>Bear (tightest fit is box)</td>
<td>4441 ms for 1 million runs</td>
<td>0.004441 ms</td>
</tr>
<tr>
<td>Dragon (tightest fit is box)</td>
<td>4708 ms for 1 million runs</td>
<td>0.004708 ms</td>
</tr>
</tbody>
</table>

Figure 6-3: Five representative models, each with the time taken to access and transform the tightest fit bounding structure.

### 6.2.4 Performance of intersection tests

Each intersection test was run over 100,000 times and the total time taken was measured. Thus precise values for individual intersection tests were obtained.

Figure 6-4 describes the results for the six types of pairwise intersection tests.

<table>
<thead>
<tr>
<th>Type of intersection</th>
<th>Raw results</th>
<th>Time per intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere-Sphere</td>
<td>319 ms for 1 million runs</td>
<td>0.000319 ms</td>
</tr>
<tr>
<td>Collision Type</td>
<td>Time for Runs</td>
<td>Time per Collision</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Capsule-Capsule</td>
<td>3608 ms for 1 million runs</td>
<td>0.003608 ms</td>
</tr>
<tr>
<td>Box-Box</td>
<td>11,717 ms for 100,000 runs</td>
<td>0.11717 ms</td>
</tr>
<tr>
<td>Sphere-Capsule</td>
<td>2136 ms for 1 million runs</td>
<td>0.002136 ms</td>
</tr>
<tr>
<td>Sphere-Box</td>
<td>2191 ms for 1 million runs</td>
<td>0.002191 ms</td>
</tr>
<tr>
<td>Capsule-Box</td>
<td>4275 ms for 100,000 runs</td>
<td>0.04275 ms</td>
</tr>
</tbody>
</table>

Figure 6-4: Performance of pairwise intersection tests

6.2.5 Performance of overall system

The same setup described in section 6.1 was used to test the performance of the TightestFitCollider overall. For 100 spheres, which is 10,000 collisions, collision checking took 78 milliseconds. The current system, in comparison, takes 30 milliseconds. The additional time comes from accessing the stored spheres and transforming them using matrices and vectors. For the current system, these transformations are built-in, since all bounding structures are spheres. For 100 bears, which is again 10,000 collisions, the TightestFitCollider takes 1300 milliseconds, or about 0.13 milliseconds per collision. This number matches up to the time taken to perform a box-box intersection plus two box transformations, so the test results are consistent.
6.3 HierarchicalCollider

6.3.1 Performance of construction

The same twelve models discussed in section 6.2 were tested to find how long it takes to construct their bounding box hierarchies. For these performance measures, the maximum depth of the bounding box hierarchy was set to ten, and nodes with five or fewer vertices were not split further. The time it takes to construct the bounding box hierarchy is proportional to the number of vertices in the model and the distribution of those vertices. The times are shown in Figure 6-5 below.

<table>
<thead>
<tr>
<th>Model</th>
<th>Time to construct hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>0.536 ms</td>
</tr>
<tr>
<td>Cube</td>
<td>0.066 ms</td>
</tr>
<tr>
<td>Cylinder</td>
<td>1.685 ms</td>
</tr>
<tr>
<td>Bear</td>
<td>3.722 ms</td>
</tr>
<tr>
<td>Dragon</td>
<td>25.542 ms</td>
</tr>
<tr>
<td>Seagull</td>
<td>7.728 ms</td>
</tr>
<tr>
<td>Tree</td>
<td>3.964 ms</td>
</tr>
<tr>
<td>Kelp</td>
<td>0.359 ms</td>
</tr>
<tr>
<td>Fortress</td>
<td>3.874 ms</td>
</tr>
<tr>
<td>Spaceship1</td>
<td>3.016 ms</td>
</tr>
<tr>
<td>Fireman</td>
<td>10.479 ms</td>
</tr>
</tbody>
</table>
Figure 6-5: Twelve models and the time taken to construct their bounding box hierarchies.

The max depth of the hierarchies is set to ten and nodes with five or fewer vertices are not split further.

6.3.2 Performance of transformation

The time taken to transform a hierarchy of bounding boxes is the same as the time taken to transform a bounding box (0.003 – 0.005 ms), as discussed in section 6.2.3, multiplied by the number of boxes that need to be transformed as the hierarchy is transformed.

6.3.3 Performance of traversal

The time taken to traverse the trees is proportional to the number of nodes that are visited. For each node-node comparison, there are two box transformations and a box-box intersection test. To get an idea of the number of nodes visited, the traversal ends as soon as an intersection is found between any two leaf (or max depth) nodes. The traversal returns quickly from one part of a hierarchy tree if two nodes are found not to be intersecting. The worst-case traversal would be if all of the nodes intersected with each other right up until the leaf nodes every time. Then every single node in both hierarchies would be visited.

6.3.4 Performance of overall system

The same setup described in section 6.1 was used to measure the overall performance of the TightestFitCollider. For 10 bears or 10 spheres, which is 100
collisions, the system took 75 milliseconds. For 100 bears or 100 spheres, which is 10,000 collisions, the system took roughly 7500 milliseconds.

These values make sense for the most part. During these tests, the max depth during traversal was set to 5. Since all of the agents were colliding and the hierarchy uses a depth-first approach, there were 5 node-node checks done for each pair of agents. Thus there were 500 sets of two box transformations and a box-box intersection test. That is equal to 500 * (0.005 + 0.005 + 0.12) = 65 milliseconds for the 100 collisions case. The additional 10 milliseconds over 100 collisions is likely due to function calls in the traversal, additional calls that return early, and the checks that are in place to see whether a call can return early. This system cannot be used for the large number of collisions that were tested, but that was known from the start. The idea is to use hierarchical colliding on one or two main agents in the game. For around 10 collisions with a main character, the system would take 7.5 milliseconds, which is reasonable performance.

6.4 CombinationCollider

The overall system performance of the CombinationCollider would be somewhere between the overall system performance of the TightestFitCollider and that of the HierarchicalCollider, depending on how many agents use the hierarchical collisions. It would also depend on the number of collisions happening at once, and the types of bounding structures used in those collisions (e.g. box-box intersections are slower than sphere-sphere intersections).
7 Future work

Some of the potential areas for future work are listed below:

1. Add blocks to the StarLogo Nova interface that allow the user to specify whether an agent should use hierarchical collisions or not.

2. Add blocks to the StarLogo Nova interface that allow the user to specify a max depth for hierarchy traversal if the agent is hierarchical.

3. Render the bounding structures within the StarLogo Nova framework, so that they can be visually checked against the agents they are fitting. The rendering could also be extended to show the users the various bounding structures and allow them to choose which ones they prefer for the models they care about.

4. As an optimization feature, choose spheres over capsules and capsules over boxes if their volumes are within some factor of each other – rather than choosing the absolute best volume each time. This optimization would choose a higher percentage of bounding structures that have faster intersection tests.

5. As another optimization feature, do sphere-sphere or capsule-capsule intersections as a preliminary step before doing box-box intersection tests.

6. As suggested by Daniel Wendel, consider modifying the Ritter algorithm for finding bounding spheres, such that the center of the sphere is fixed in x and y (because models are pre-centered in these dimensions) and only updates in the z dimension. This approach may provide closer fitting spheres than the naïve algorithm.
7. Extend the hierarchical collider to use spheres or capsules and see if the performance tradeoffs are better or worse.

8. Determine how many collisions different types of games have on average. Thus performance in the average case can be known, instead of the worst-case measurements done here.

9. As suggested by Daniel Wendel, if StarLogo Nova is switched completely to HTML5 and Javascript, try using local storage to save the tightest fit bounding structures and hierarchies computed for each model. Then this data only need to be calculated once, instead of each time a game loads.
8 Conclusion

StarLogo Nova aims to make teaching and learning more fun for teachers and students alike. Teachers often use StarLogo Nova to create educational simulations. Students can learn more about computer science using StarLogo Nova’s block-based language, while having fun creating and playing their own 3D video games. In order to make the 3D games realistic and convincing, the graphics engine must support fast and accurate collision detection.

In response to a desire from users for more accurate collision detection, this thesis has explored and implemented a couple new approaches to collision detection in StarLogo Nova. Although these systems are slower than the current system, they are more accurate in detecting collisions. The TightestFitCollider provides tight fitting bounding structures for each agent. The HierarchicalCollider allows for even more precise collision detection using a hierarchy of bounding boxes. The CombinationCollider combines these two systems and provides the back-end support for allowing users to specify which agents should have more precise collision detection.

Ideas for future work have also been suggested, including the front-end support that will allow the users to interact with the newly developed collision detection systems. Other ideas for future work suggest ways to extend the collision detection systems further or potential optimizations. Overall, this thesis has sought to improve the collision detection system in StarLogo Nova, in order to make the 3D game experience realistic and fun.
9 References


