Starlogo Camera Controls and Enhancements

by

Christopher M. Cheng

S.B., C.S. M.I.T., 2010

Submitted to the Department of Electrical Engineering and Computer Science

In Partial Fulfillment of the Requirements for the Degree of

Masters of Engineering in Electrical Engineering and Computer Science

At the Massachusetts Institute of Technology

May 2011

[June 2011]

Author _____

Department of Electrical Engineering and Computer Science
May 16, 2011

Certified by _____

Eric Klopfer
Director, MIT Teacher Education Program
Thesis Supervisor

Certified by _____

Daniel Wendel
Research Associate, MIT Teacher Education Program
Thesis Co-Supervisor

Accepted by _____

Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Starlogo Camera Controls and Enhancements
by
Christopher M. Cheng

Submitted to the
Department of Electrical Engineering and Computer Science

May 16, 2011

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The purpose of this research and development project was to further develop Starlogo, a block based programming environment to allow students to create 3D games and simulations to help them learn programming. I improved the Starlogo "camera", a representation of the view within a 3D world. Starlogo has a 1st person perspective camera called "Agent Eye", a 3rd person perspective camera called "Agent View", and a customizable camera called "Aerial". Previously, the view could get obscured by the terrain and was disorienting at times making games difficult to play. The cameras now automatically move around obscuring terrain, have a larger field of vision, and move smoothly making games easier to control and fun to play. Users can also manually control the cameras and have a full range of vision within Starlogo worlds. These recent camera developments will be used in the next release of Starlogo and will greatly improve gameplay.

Thesis Supervisor: Eric Klopfer
Title: Director, MIT Teacher Education Program

Table of Contents:

Section 1: Introduction

StarLogo is a block-based programming system used to lower the entrance barrier to programming. The Scheller Teacher Education Program (STEP) is developing and using StarLogo in high school classrooms to educate students in addition to help students become interested in Computer Science.

StarLogo accomplishes this goal by helping students to program 3D simulations and games. To program a game or simulation using StarLogo, the user must:

1. Use puzzle-piece-like blocks to program the behaviors of the agents, the characters or interacting objects used in the game
2. Customize the terrain, the 3D world where the characters interact

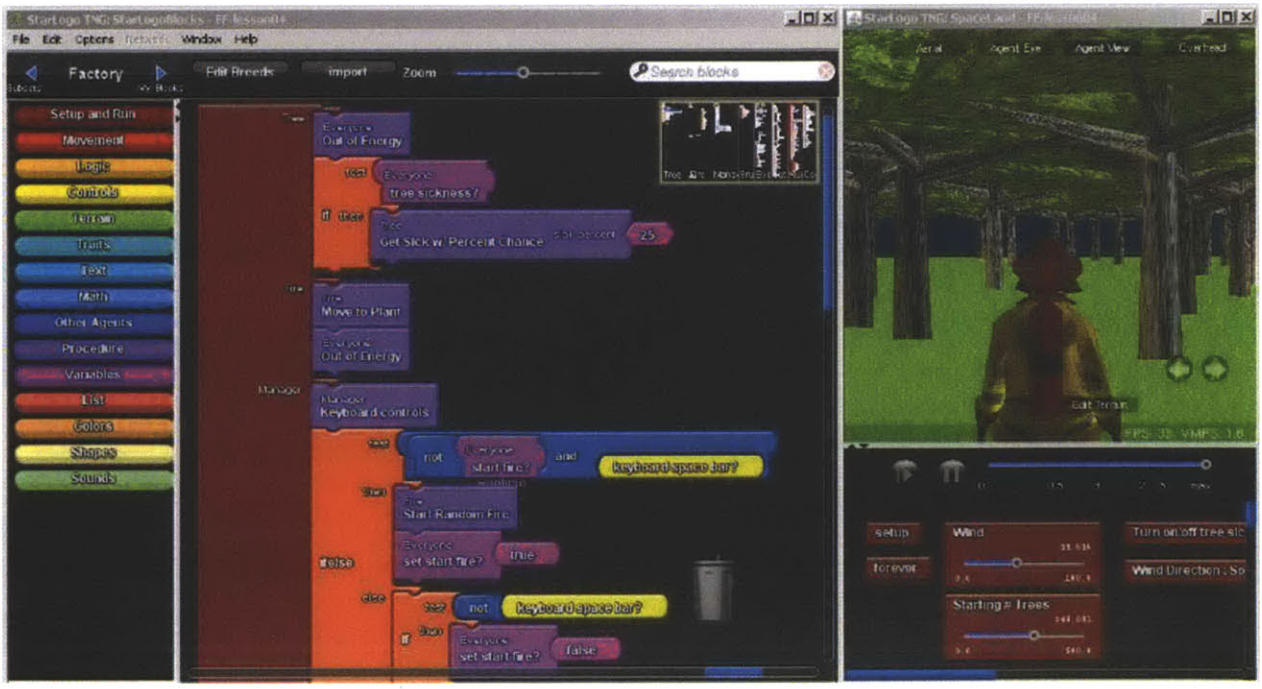


Figure 1: An example screenshot of the StarLogo Interface.

After development, the user can play a fully functional game that he/she created. The projects can also be sent in a custom-designed file format so people can play games that others created. Being able to develop a fun playable game that is aesthetically pleasing while learning programming is central to the aim of Starlogo, since the aim is not only to educate the user, but to help the user become enthusiastic about computer science.

Section 2: Recent Starlogo Developments

Before discussing more about this project's developments, it's important to discuss the recent developments of Starlogo. Starlogo terrains originally existed in 2D as a grid of "patches" where a patch is a unit square of the grid that an agent can stand on. In this environment, the user could color the different patches and place agents anywhere on the terrain to customize his/her game. Eventually, in order to greatly improve the aesthetic appeal of the games, Starlogo The Next Generation (TNG) was created. The terrains still exist as a grid of patches but each patch now has a height. The terrains aren't quite in 3D however, since patches can't float in mid-air; patches are always connected either with walls or slopes depending on the terrain. Figure 2 shows an example of a Starlogo terrain, where the terrain exists in patches and the user can modify the patches in different ways including creating a mound, vertically raising the terrain, or coloring the different patches.

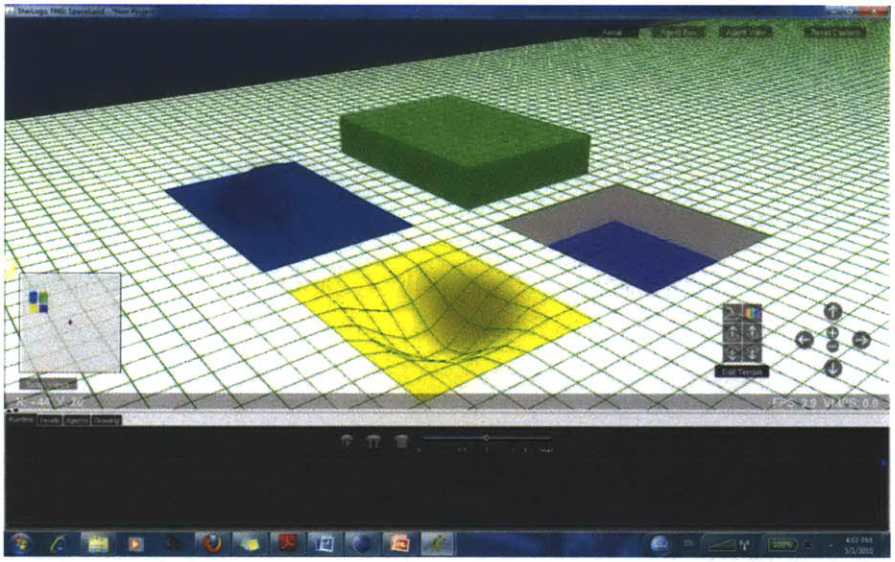


Figure 2: Screenshot showing a Starlogo Terrain. The patches can be different colors. This terrain highlights some of the different modifications a user can make to the terrain, each shown in a different patch color.

These new terrains give many more options for customizing Starlogo games. The development of new terrains also caused the need for many changes to the terrain editing tools in addition to blocks so the agents can interact with these new terrains. Papers such as those written by Daniel Wendel and Mark Burroughs talk about the transition from 2D terrains to 3D terrains and the tools they built including the 3D terrain editor and the mound creator, which were created in order to easily customize these new terrains. For my previous project, I continued development on terrain editor tools to further increase the possibilities for the terrain and make the games more fun to play. Specifically, I designed textures for the terrain so

a user could paint within patches and place arbitrary images down onto the terrain. More details can be referenced in that paper.

The goal of this project is to continue developments of Starlogo to adapt to the new terrains to make the user's experience of creating games and playing them more enjoyable. To figure out additional changes to be made, I tried playing a set of about 50 Starlogo games made by students who used these new terrains. After playing the games, the feature that stood out as needing the most improvement was the camera, since it clashed heavily with the 3D terrains and greatly hindered the gameplay of the Starlogo games, which will be explained in section 4. Before talking about the details of both the problems with the camera and the solution I developed, it's important to talk about other examples of cameras used in gaming and what constitutes a good camera.

Section 3: Background Research

Before immediately launching into the details of what my project intends to solve, it's important to do some background research to shed some insight on key details about what constitutes a good camera in a 3D world.

Camera manipulation is necessary in any video game that involves moving around in a 3D environment. One game worth looking at is World of Warcraft, a massively multiplayer online game where a player can control an avatar in a 3D world completing various quests. This game does a very good job with a third-person camera perspective in a number of different scenarios including walking in a wide open area, going into houses or small caves, and going up hills.

After watching how the game works, it seems that the camera defaults to a fair distance behind the character, raised slightly above the character, and slightly angled downwards. Positioning of the camera far enough behind the character maintains a good field of vision for the player so the player can see into the distance to know what is happening in the game. Raising the camera allows the player's vision of the distance to not be occluded by their own character so the player can see ahead without having to turn. And angling the camera downwards gives the player a good view of their own character to see what is directly in front and in back of their character.

The World of Warcraft camera also does a good job maintaining a good amount of vision in scenarios trickier than walking in a wide-open area. When approaching hills or valleys, the camera's height and angle will change in order to maintain a good view of what is ahead of the character, even if the character is looking directly at the bottom of the hill or valley. If the

character moves into an enclosed location such as a tunnel or a house, the camera stays as far back as it can while staying in front of any wall or ceiling that could be in the way, which prevents the camera from getting to a position where a wall could occlude the player's view of the character. One last thing the camera does is smoothly move between positions so the player can focus on the game itself without being distracted by the camera. The camera used in World of Warcraft highlights many important features of a third-person perspective camera so the player always has a clear view of their character and what else is happening in the game. Some example screenshots of World of Warcraft are displayed in figure 3.

Figure 3: Some example screenshots of World of Warcraft. Its third-person perspective gives a great field of vision to the player in a wide range of different situations.

Another game worth looking at is Halo for the Xbox 360. Halo uses both third and first person perspectives throughout the game. Its third person perspective has many of the nice features that World of Warcraft's did, although Halo's first person perspective is worth looking at since players use the first person perspective most of the time. The first person perspective consists of the view as seen from the controlled character and the gun the player is currently holding. Even though the gun moves to show that the character is walking, the camera moves smoothly so the player has a smooth view of the world and where he/she is going. The camera also stays level with the terrain and doesn't tilt even when the character is on a slope in order to not disorient the player. The camera used in Halo highlights important things to take into consideration when designing a first-person perspective camera including smoothness, tilt, and control. Some example screenshots of Halo are displayed in figure 4.

Figure 4: Some example screenshots of Halo. Its first person perspective is smooth and controllable in a variety of situations.

One last game worth examining is Pikmin, a GameCube game involving moving through maze-like environments with creatures called Pikmin to help you accomplish different tasks. Along with a third-person perspective, it also allows the player to switch to an aerial camera, which gives the player a top-down view of the character. This view is especially helpful when moving through mazes, since it gives the player a view of a larger surrounding area, even when the character is unable to see certain parts of the maze. In the game, there are no ceilings or floating objects that occlude the view of the character. When the aerial camera actively tracks the character, the player always has a clear view of the character and a large radius of the surrounding area. At certain points, the game even automatically switches to the aerial camera when the third-person camera would clearly be occluded, which allows the player the convenience of not having to constantly need to control the camera. One last feature of this game's camera is that it allows easy switching of cameras, since even though the aerial view is useful at certain points, at other points it's not as useful since the player can't get a good view of what is happening near the character. The camera used in Pikmin highlights more interesting points about cameras including how an aerial camera including can give a much larger field of view for the player in addition to how switching between cameras is often necessary for the gameplay of certain games. Some example screenshots of Pikmin are shown in figure 5.

From the examples, it's clear that there are multiple cameras that are useful for 3D gaming and simulations. All of the cameras focus on giving the player a smooth and large field of vision so the player receives the information he/she needs for the game to create good gameplay. In the next section I will discuss the cameras used in Starlogo before this project, and the issues that created difficulty in playing Starlogo games.

Figure 5: Some example screenshots of Pikmin. Its zoomed out aerial perspectives allow the player to get a good view of the surrounding area even when the character is inside maze-like structures

Section 4: Previous Camera Issues

The main aim of my project is to improve the cameras used to view the Starlogo 3D terrains within the Starlogo games and simulations. Starlogo provides the user with "Agent View", "Agent Eye", and "Aerial" cameras allowing the user to view the characters or "agents" from a close distance behind, view the first person perspective of an agent, and view the terrain from a customizable angle respectively. These cameras can be accessed on the Terrain Editor window or through a set of camera blocks that switch between cameras.

Although the cameras function properly, the Agent View and Agent Eye cameras have issues that hinder the gameplay of the games. Starlogo's "Agent View" camera follows an agent at a fixed distance behind so the player can see the agent being controlled in addition to the rest of the world. However, by placing the camera directly behind the agent, the player's view of the world is partially occluded by the agent itself. This means the player needs to keep turning in order to see what is directly in front, which makes controlling an agent in a game very inconvenient as shown in figure 6.

Another problem is that the camera doesn't ever move from its relative position behind the agent. If a terrain feature like a mound or a wall or another agent in the game is positioned between the camera and the controlled agent, the player will no longer be able to view his/her agent which can make the game very frustrating and even unfeasible to play. Since terrain editing is a central feature to how Starlogo games can be customized, this is a significant problem. An example of how the terrain can occlude the view of the controlled agent is shown in the two images of figure 7.

Figure 6: Example of Agent View. Since the camera is directly behind the agent, the user cannot see what is directly in front of the agent.

Figure 7: Example images showing how terrain can occlude the Agent View camera. The left image is the aerial camera showing that a hill is directly behind the agent. The right image is the agent view camera in the same scene, where the camera is placed behind the hill and the view is occluded.

One last problem with the Agent View camera is that sometimes the camera appears to stutter every few frames when using Starlogo. This is not simply due to the computations taking too long, but actually has to do with the smoothness calculations, which will be explained more in section 8. The lack of smoothness is not only disruptive to gameplay, but

makes any created game look unpolished. The problems of the Agent View camera need to be fixed, since fixing the problems will allow the user to always see their agent and know what else is happening in the game, making gameplay much easier to control and much more enjoyable.

Like the Agent View camera, the Agent Eye camera also has issues that hinder the gameplay of Starlogo games. The "Agent Eye" camera gives the player a first person perspective from the agent, so in a sense the camera is placed directly at the agent's eyes. One problem with the camera is that the field of view is too small. Since the placement of this camera is in front of the placement of the Agent View camera, the player's field of view is reduced and forces the player to turn more in order to see things in the peripheral. This is shown in figure 8.

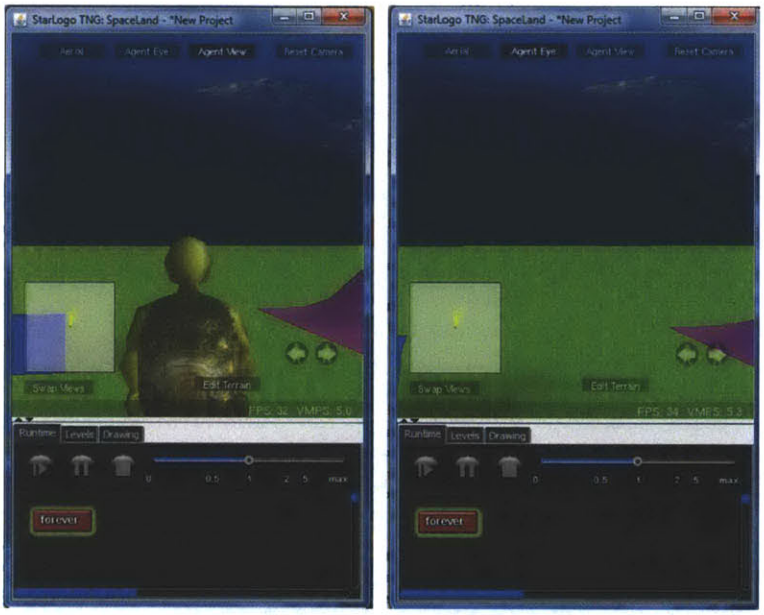


Figure 8: Screenshots showing the loss of field of view using the Agent Eye camera. The left shows the view with Agent View, and the right uses Agent Eye.

Another problem with this camera is that allows for too much tilt. If the agent is positioned on terrain that isn't flat such as on a mound or a valley, the camera will be tilted to reflect the slope that the agent is standing on. Depending on the agent's direction when standing on slope, the camera might be tilted horizontally or vertically as shown in figures 9 and 10 respectively.

While camera tilting is desired in some games for being realistic, too much tilt can cause problems. Having too much horizontal tilt can be very disorienting making it difficult to traverse the terrain and aim properly. Too much vertical tilt can lead to undesirable camera angles, since the agent might get to a point where it is looking directly down at the ground or

directly up at the sky. These camera views give no information about the rest of the world and detract from gameplay. The problems caused by tilt need to be addressed even if it means removing the tilt completely.

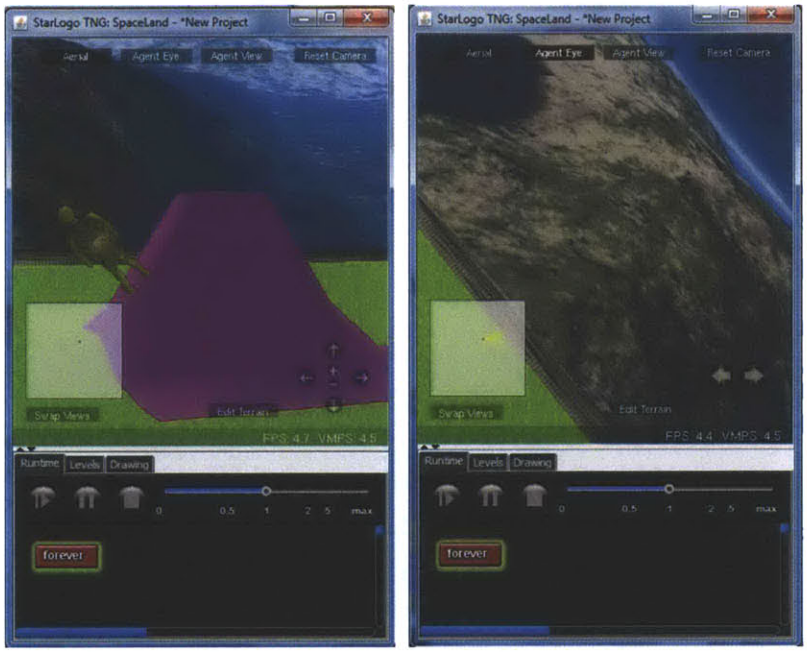


Figure 9: Screenshots showing the tilted view when standing on a hill. The left shows where the agent is standing in the world, and the right uses Agent Eye from the same position.

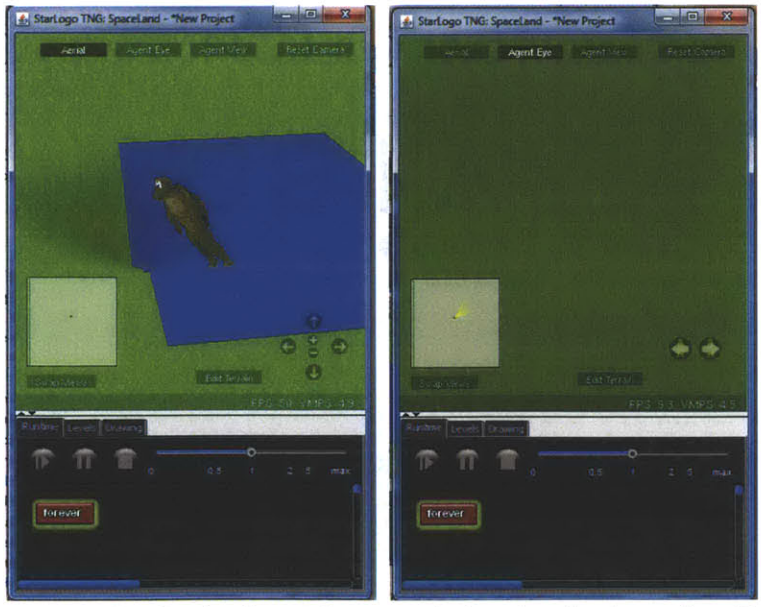


Figure 10: Screenshots showing the view looking at the ground when standing in a valley. The left shows the agent standing in the valley, and the right uses Agent Eye from the same position. The agent looks straight at the other side of the valley.

It should be noted that the Agent View camera will also tilt vertically depending on the agent's position. The Agent View camera does not suffer from the same problems as the Agent Eye camera, since the field of view is larger. Even when the agent view camera is looking downwards or upwards, the user still gets enough information about what is happening in the game.

One last problem with the Agent Eye camera is that it suffers from the same stuttering problems that the Agent View camera has. The lack of smooth motion is very disruptive to gameplay and I will explore more of these calculations in section 8.

The issues with the Agent Eye camera in addition to the Agent View camera need to be addressed in order to greatly improve the gameplay of Starlogo games. In addition to fixing the current issues, there are additional features that I wanted to add for the cameras that were either requested by the students, or thought of during the project, which I will discuss next.

Section 5: Additional Requested Improvements

In addition to fixing problems with the existing cameras, I also wanted to provide additional enhancements to the cameras to further improve Starlogo gameplay. Since Starlogo is currently being used by students in classrooms, the Teacher Education Program has received feedback from the students including feature requests. Surprisingly, one of the most common feature requests regarding the camera is to have a 2nd person view camera. At first this seemed counterintuitive since this view only allows the viewer to see his/her agent, and not the surrounding area. However, most of the students were encouraged to include some storyline along with their game creating the need to view the front side of their agent. Currently, this is accomplished by creating a separate agent for camera purposes and positioning this agent in front of the desired agent. While this is functional, it seems unnecessary to have to create another agent for the camera. Since this feature is commonly used, it would be helpful to add this feature, which will be discussed more in section 9.

In addition to the feature request, I also wanted to create an overhead view that could track agents. The usefulness of this view is described in section 3, where an overhead view can be especially useful in some games such as those that involve mazes. From the previously existing code, the Aerial view comes the closest to accomplishing this. The "Aerial" view is a customizable view of the terrain where the user can use the mouse to rotate the camera around the terrain, pan the camera back and forth, and zoom in and out. However, while it is customizable, it is not very feasible to use in some games when controlling an agent. The issue

with using the Aerial camera during a game is that it does not track individual agents, so if the agent moves off of the screen, the user must manually move this camera to get the agent in view and then resume controlling the agent. This is a problem not only for the inconvenience of having to manually move the camera with the mouse, but it's also a problem in that the user may miss something in the game while the controlled agent is off screen. This is shown in figure 11.

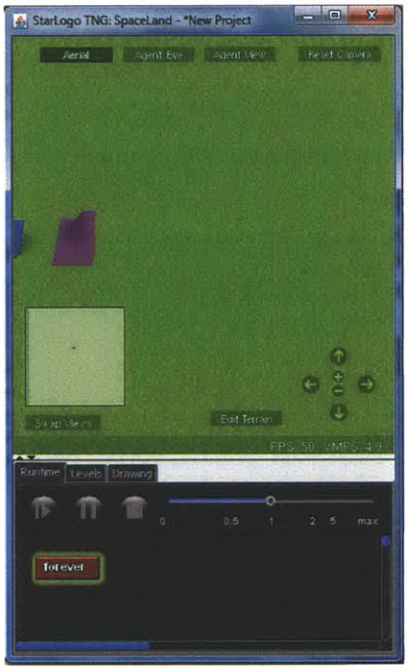


Figure 11: Screenshot showing how the Aerial camera placed so the controlled agent (on the blue terrain) cannot be seen.

In order to have the agent stay on screen at all times, the camera would have to be zoomed out far enough such that the user could see the whole terrain. If there are lots of terrain features like hills or walls, the only way to ensure that the user could see the agent is if the camera is positioned overhead due to the terrain implementation mentioned in section 2. The problem is that during the game the player can't get an accurate view of the agent and what is happening in its near vicinity. This is shown in figure 12. It would be nice to have some sort of overhead view that tracks the agent and stays close enough to the agent so the user can see what is happening in the game.

One last feature that I want to add is to make the camera views customizable to give a greater range of vision. It would be desirable to modify the camera views such as looking to the left or right to give an increased field of vision while still following the agent. This would enhance the 1st person and 3rd person views by giving the user more control over what they can see and would enhance the gameplay of Starlogo games. More of this will be talked about in

section 9. In the next section however, I will first talk about the modifications made to address the issues with the cameras before talking about additional enhancements.

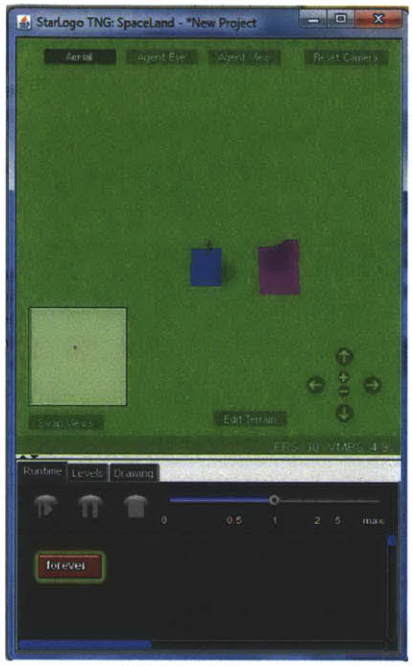


Figure 12: Screenshot showing how the Aerial camera usually needs to be positioned very far away to get a good range of vision, but it's so far away that it's difficult to view the agent (on the blue terrain)

Section 6: Agent View Camera

The 3rd person perspective Agent View camera underwent the most changes throughout the entire project. Not only was the field of view increased, but the camera will rotate upwards to give more of an overhead view if there is terrain that would occlude the view of the character. The movements for this camera were smoothened which will be discussed in section 8 and manual control for the camera was created which will be discussed in section 9. I will begin this section by discussing how the previous code used to work, and then proceed to the recent changes.

Section 6.1: Previous Code for Agent View

The implementation for the movement of all of the cameras is within the method:

updateTurtleCameras(double deltaTime)

within TorusWorld.java. This method is called whenever we're redrawing the terrain and calculates the new position and orientation of the camera Starlogo is currently using. The input deltaTime, is how much time in seconds that have passed since the previous time this method has been called. This method will behave differently depending on which camera Starlogo is using. The camera that is currently being used is stored as the variable

SLCameras.currentCamera

which is an int that when running Starlogo, may take on the values:

SLCameras.OVER_THE_SHOULDER_CAMERA – for Agent View
SLCameras.TURTLE_EYE_CAMERA – for Agent Eye
SLCameras.PERSPECTIVE_CAMERA – for Aerial

SLCameras is a static class that contains a reference to an instance of Camera, the class containing the mathematical data of the camera. It also contains which camera is being used, as well as other methods for using the camera when drawing the Starlogo terrain using OpenGL.

Camera is a class that contains the information about the position and orientation of the camera in terms of three vectors of three floats each. The three vectors contain the position of the camera, the direction the camera is facing, and which direction is upwards. Note that the "upwards" direction for the agent is analogous to the normal vector of the terrain the agent is standing on; it is not which direction is "upwards" globally. The camera also contains some other methods including those for useful transformations such as translations, vertical rotations, and horizontal rotations. Although Camera and SLCameras contain the data for the camera, the method updateTurtleCameras is where the camera position and orientation are determined, which is what I want to modify.

When Starlogo is using the Agent View camera this method sets the camera's position and orientation in the following code segment within updateTurtleCameras:

```
pos.set(0, turtleViewPosition.boundingCyl.top * 0.9f, 0);
dir.set(0, 0, -1);
up.set(0, 1, 0);

turtleViewPosition.localCoordSys.localToGlobal(pos);
turtleViewPosition.localCoordSys.localToGlobalDirection(dir);
turtleViewPosition.localCoordSys.localToGlobalDirection(up);
```

In the code segment, the camera's direction and up vectors are set to be the direction and up vectors of the agent respectively in coordinates for the Starlogo terrain. The position is set in a similar way to about 90% of the height of the agent, which attempts to be at about "eye level".

More specifically, this code segment uses turtleViewPosition to convert from the agent's local coordinate system to the global coordinate system of the Starlogo world. In the agent's local coordinate system, the agent faces in the negative z direction with the upwards direction pointing in the positive y direction. In the example above, the localToGlobalDirection method takes these directions and converts them to global coordinates. In the global coordinate system currently the positive y direction is upwards and the positive x and z directions are east and south on the terrain respectively. The center of the center patch at ground level is the origin with each patch length being three units long. A similar calculation process is done for the position.

The method updateTurtleCameras then translated the camera backwards relative to the agent using the method:

camera.translate(float dx, float dy, float dz)

This will translate the camera relative to its own orientation in the metric of world coordinates. Since the camera is already pointing in the direction of the agent, using this method with dx = 0, dy = 0, dz = amount to translate, will translate the camera backwards as expected.

This was all of the pre-existing code that set the position and orientation of the agent view camera. This information is then passed to the CameraSmoother class which will be discussed more in section 8. Note that none of these calculations take any of the terrain into account, which can lead to terrain coming between the camera and the agent which disrupts gameplay.

Section 6.2: Recent Changes to Agent View

In order to improve the Agent View camera, I wanted to improve the field of vision and make the camera rotate vertically around obscuring terrain to make sure the user always has a clear view of his/her agent as well as the surrounding terrain.

Section 6.2.1: Increasing the Field of Vision

To increase the field of vision, I translated the camera farther back and upwards from its previous position relative to the agent. To do this, I kept the code from the previous section which positions the camera at the agent's "eye level", and then I used both the translate method described within section 6.1 and the translateGlobal method which translates the camera according to global Starlogo terrain coordinates instead of the local coordinates of the camera. After some experimentation, I found that translating the camera 2.9 units globally upwards from the "eye level" lets the user see far enough ahead of the agent. I also found that translating the camera a distance of 30/sin(angle between camera's direction and x-z plane) behind the agent provided a good field of vision around the agent. Note that the translation backwards is in the direction of the camera. The formula for the backwards translation will keep the camera at a constant horizontal distance of 30 behind the agent as shown in figure 13. Keeping the camera at a constant horizontal distance behind the agent will be important later in discussing the motion of the camera in section 6.2.3.

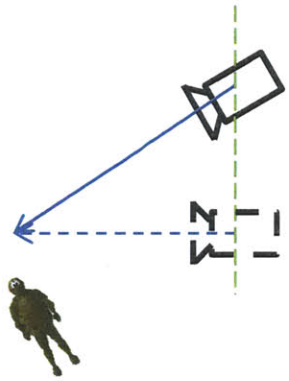


Figure 13: Figure showing how the camera stays at a constant global horizontal distance behind the agent. The dotted camera represents the position of the camera if the agent were standing vertically. If the agent is tilted, the camera translates backwards relative to the agent a farther distance to keep the same global horizontal distance.

Figure 14 shows the new position compared to the old position: how the new position isn't obscured as much by the agent, and how the user is able to see more of the Starlogo terrain. Initially I tried making these numbers proportional to the size of the agent, although I found that larger agents didn't need the camera translated farther back. With larger agents it was still possible to see the nearby terrain while not being so far away that the user has trouble seeing the immediate surroundings of the agent. One might think this wouldn't hold if an agent was large enough although if an agent is much larger than the typical size of a few patches, the agent would be a significant size of the whole Starlogo terrain and wouldn't be very useful to use from a 3rd person perspective.

Figure 14: Screenshots showing side by side comparison of before and after placement of the camera relative to the agent. Note how the field of vision is much larger on the right one.

Section 6.2.2: Checking for Obscuring Terrain

The other main change I made to the Agent View camera was to make it rotate above obscuring terrain. The basic approach I took to accomplish this goal was to search along the line from the camera position to a point on the agent and check a sample of points along the terrain for terrain that would obscure the view of the agent. This is illustrated in figure 15 that gives a top view of the terrain showing the straight line path from the camera to the agent. The path is dotted to illustrate the evenly spaced points along the terrain that are checked to see if the view of the agent is obscured.

However, this problem is not that simple since the agent has a width; there's not simply one point on the agent we want to see. Instead of checking that a line of vision isn't blocked, we want to check whether a tunnel of vision isn't blocked where the tunnel has the width of the agent and is pointing from the camera to the agent. This is illustrated in figure 16. Checking a tunnel of points is important since it increases the chances of finding obscuring terrain if there is any. This is also illustrated in figure 16. The patches that are shaded in green are patches that potentially could obscure the view of the agent and might not be found on the points from the path directly from the camera to the agent. The new camera will check points along the edges of the tunnel in addition to the direct path from the camera to the agent. These points like before are indicated by the placement of the red dashes in figure 16.

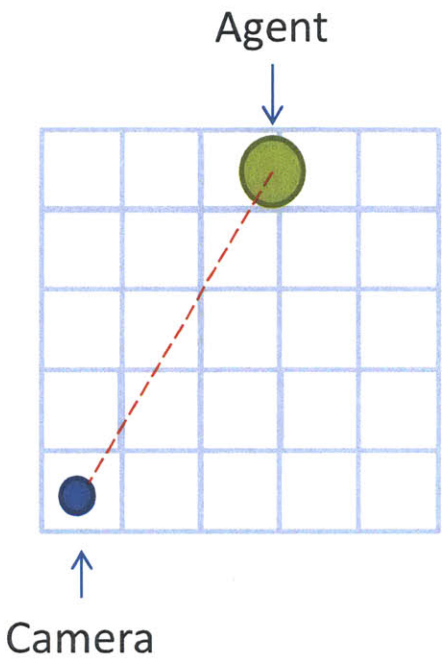


Figure 15: Figure showing a top view of the terrain and the path from the camera to the agent. The red dashed line represents the path, and the red dashes themselves represent the evenly spaced points along the terrain that we can check on the path.

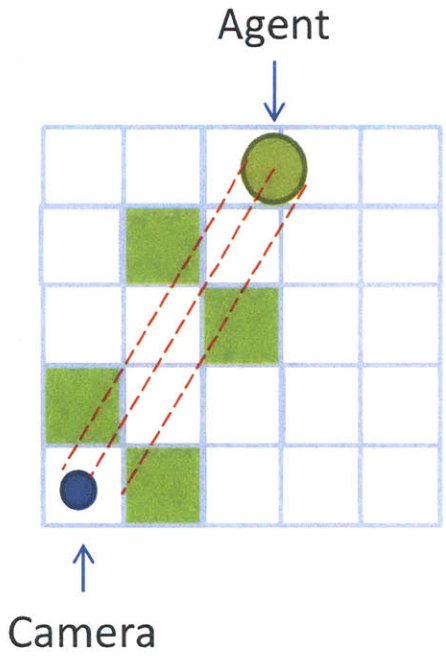


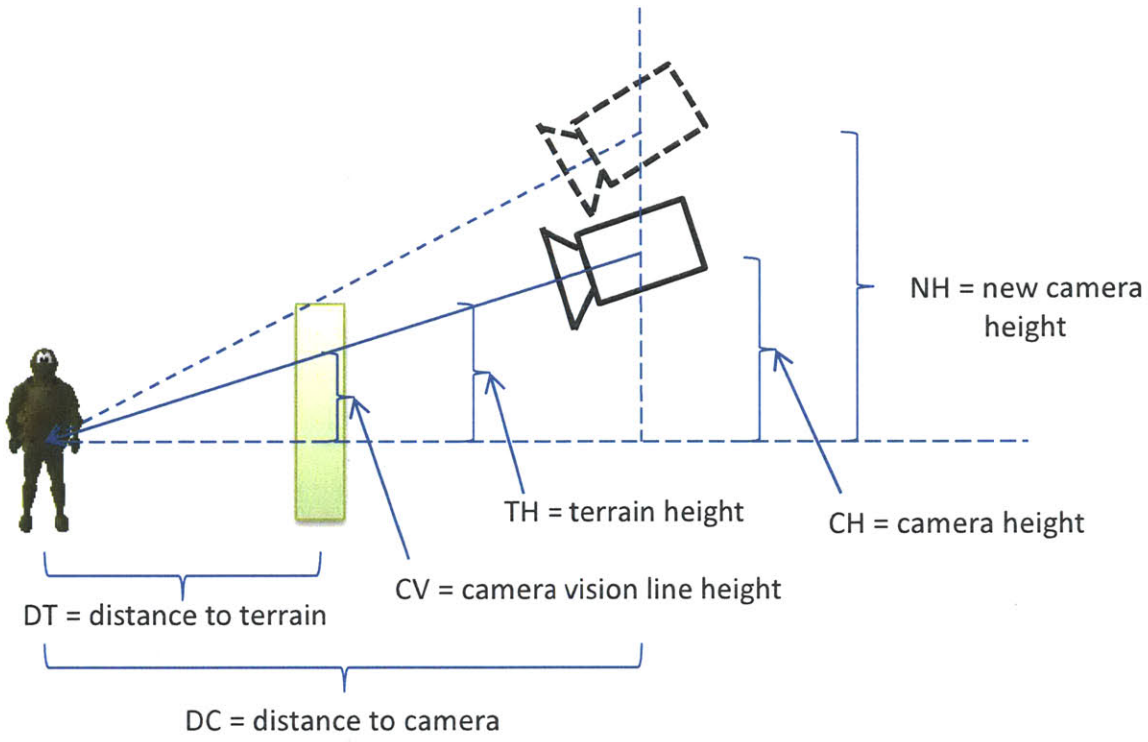
Figure 16: Figure showing top view of the terrain and the line of sight tunnel from the camera to the agent. The red dashes represent the points along the terrain we want to check for terrain obscuring the view of the agent. Patches shaded in green are patches that potentially could obscure the agent, but might not be found by the center red line, the direct path from the camera to the agent.

One additional challenge in searching for terrain that would obscure the agent is that the calculations need to run in an efficient amount of time. If the calculations take too long, there will be a large delay between the user input and the screen display since the screen display needs to wait for the camera placement calculations to finish. To simplify the calculations of choosing which points on the terrain, I chose 25 evenly spaced points along the path to the agent and 25 points along each edge of the tunnel, where the tunnel edges are shown in figure 16. Checking these 75 points should find most of the obscuring terrain. There may be cases where the 75 points miss some obscuring terrain especially if the agent size is very large. However, for reasonably sized agents the majority of cases should be handled properly while taking a short amount of time to perform the calculations.

Now that we've established which points on the terrain we want to check for obscuring terrain, we must discuss how we check for obscuring terrain at a given point. The line going from the camera position to the agent is downward sloping and ends at a point on the agent that we want to make sure is not obscured by the terrain. I picked this point to be approximately at the waist level of the agent, more specifically at 40% of the agent's height. The tunnel edges are the same line as the center line, only the tunnel edges are translated horizontally to end at the sides of the agent as shown in figure 16. At any point that we check along one of these lines, if the terrain height is higher than the height of the line, we need to raise the camera upwards to look around this terrain. We can trace a line from the point on the agent to the position of the camera and calculate the height that the camera needs to be raised to in order to see the agent above this terrain. A diagram illustrating this along with the calculations performed is shown in figure 17.

It should be noted however, that I made a slight modification to this method. The point checked could be anywhere along the patch and raising it simply based on the height of the terrain might not be enough as shown in figure 18. To compensate for this while keeping calculations simple, I added a slight offset such that the camera needs to be raised to the height of the terrain plus the offset. Since a change in height becomes more pronounced when checking points closer to the agent, this offset decreases as the points are closer and closer to the agent, as shown in figure 18.

The algorithm proceeds through all 75 points and calculates at each point the necessary height of the camera to have a view of the agent which is not blocked by terrain. While going through all 75 points, the algorithm keeps track of the maximum height to move the camera to before modifying the camera. After the algorithm finds the height that the camera needs to be at, the camera's new position and orientation will be calculated, which I will discuss next.



Calculations performed for each point on the terrain:

If TH > CV

NH/DC = TH/DT

NH = TH* DC/DT

Figure 17: Figure showing how obscuring terrain is found relative to the agent and camera and how the algorithm calculates the new height of the camera. The algorithm traces along the solid blue line from the camera to the agent. If the terrain height is larger than the path, we want to move the camera upwards to look around this terrain. This is indicated by the slanted dotted blue path from the agent to the translated camera that looks around the terrain.

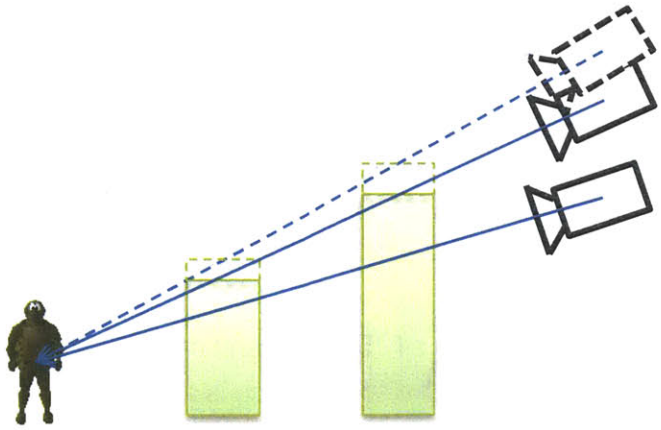


Figure 18: Figure showing the additional adjustment I made when checking terrain height. The lowest camera is the original position of the camera. The next lowest is the translated camera without the terrain offset. Note that its view is still occluded by the terrain even though we translated according to the height of the terrain. The dotted camera represents the position with the offset taken into consideration. Notice how changes in height become more pronounced when closer to the agent, so the terrain offset shrinks as the algorithm approaches the agent.

Section: 6.2.3: Camera repositioning

For moving the camera, I originally tried vertically translating the camera and rotating the camera downwards to view the agent as shown in figure 17. Although this will allow the user to see the agent, there's no bound on how high the camera translates. In particular, if there is a large terrain obstacle right next to the agent, the camera would have to translate extremely high to see around this. In this circumstance, even though technically the user's view of the agent is not obscured, the camera is positioned so far away from the agent that the user can no longer see the agent or the terrain. To prevent this, I capped the height of the camera so it can't rise above a certain distance above the agent and to ensure the user has a satisfactory view of the terrain. After some experimentation I set this value at 60, which is equivalent to 20 patch lengths above the height of the agent.

However, although capping the height ensures that the user can see the terrain, this may cause the terrain to obscure the view if the needed height was higher than 60 above the agent. To ensure the user can see the agent in this circumstance, we can place the camera at a position 60 above the agent looking straight down at the agent. Since floating terrain cannot exist in Starlogo as mentioned in Section 2, placing the camera directly above the agent ensures that terrain cannot obscure the camera's view.

Capping the maximum height of the camera position as well as positioning the camera overhead ensures we can always see the agent. However, one last problem with this camera repositioning algorithm is that the transition from vertical translation to the overhead position is disjoint. As the user is moving around in his/her game, the camera might switch back and forth between the two which can be very disorienting for the user. To keep camera motions smooth, instead of vertical translation the camera should move in an elliptical path from a horizontal viewing position to the overhead position as shown in figure 19. Figure 19 shows where the camera should be positioned based on its height regardless of whether it was modified by obscuring terrain. Note that this also includes the height of the camera due to the translations mentioned in section 6.2.1. Consistently following this path will create a smooth motion of the camera, which enhances Starlogo gameplay.

Note that an elliptical path is only used for camera positions above the height of the agent as shown in figure 19. After checking the terrain points, the camera might be positioned below the agent's height level such as in the case that the agent is standing on a hill looking upwards. The path showed in figure 19 shows that the camera will use vertical translation only for positions below the agent's height level. The reason why we don't use an elliptical path for positions below the agent's height is that an elliptical path would move the camera closer to

the agent. Moving closer to the agent would reduce the field of view, and might potentially move it in front of obscuring terrain. This vertical motion below the agent's height level keeps the horizontal distance constant at 30, which was mentioned in section 6.2.1.

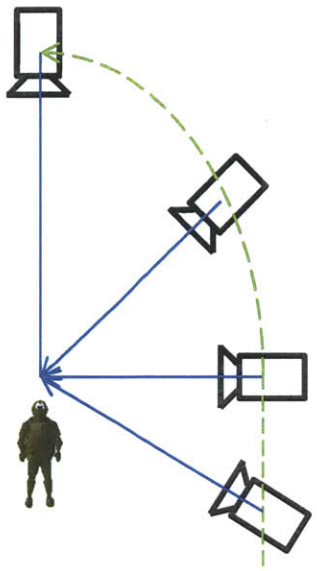


Figure 19: Figure showing elliptical motion of the camera when moving the camera to avoid obscuring terrain. Note that the elliptical motion only applies when moving to a height above the point the camera looks at. For movement below this height, vertical translation is used.

Thus, given the height of the camera, we can calculate the camera's position along the path in figure 19. For positions below the agent height, the camera simply translates upwards. For positions above the maximum height of the elliptical path, we can position the camera at the top of the ellipse. For positions on the elliptical path, we can define the elliptical path in terms of two dimensions: its height in the y dimension, and the distance to the agent in a direction in the x and z plane. The center of the elliptical path is the point the camera is looking at before translation; it's the point that is 2.9 units globally upwards from the agent which was described in section 6.2.1. Using the value for the new camera height that was calculated using points on the terrain, we can use the formula for an ellipse to calculate the new distance to the agent and then move the camera to this closer distance. The formula is displayed in figure 20.

$$\frac{horizontalDistance^2}{(initialHorizontalDistance = 30)^2} + \frac{height^2}{(maxAltitude = 60)^2} = 1$$

$$horizontalDistance = 30 * \sqrt{1 - \frac{height^2}{60^2}}$$

Figure 20: Figure showing the formula used to calculate the new horizontal distance given the height of the camera along the elliptical path.

In this section we have thus taken the necessary camera height found from the 75 terrain points and calculated the new position of the camera along a set path. The last step is to orient the camera which I will discuss next.

Section 6.2.4 Camera Reorienting

After repositioning the camera, we need to make sure that the camera is oriented such that the player can see the agent. There are two facts that will guide this process. The first is that the camera should always be looking at the same point as shown in figure 19. The second fact to note is that the camera repositioning process described in sections 6.2.2 and 6.2.3 will only raise the camera's height; thus, in order to reorient the camera, only a vertical rotation about its current position is needed. To perform this vertical rotation we can use the method:

Camera.rotateVerticallyAround(Vector3f point, float angle)

Like its name implies, this method will rotate vertically around the specified point by an amount described by the specified angle in radians. We need to use this method instead of simply setting the camera's direction since we need to rotate the camera's up vector as well or else the perspective will not display correctly. To find the angle to rotate by, we can use the formula shown in figure 21 for the angle between two vectors A and B.

$$cos^{-}1\frac{A \cdot B}{||A|| * ||B||} = angleBetweenVectors$$

Figure 21: Formula to calculate the angle between two vectors

We can rotate the camera's orientation downwards by this angle. This will cause the camera's direction to point towards the point shown in figure 19 and the up vector to stay orthogonal to the direction vector. After reorientation, the camera should now be positioned and reoriented around the terrain giving the player a good view of the agent. Some example screenshots of the improved Agent Camera are shown in figure 22 and figure 23.

Figure 22: Screenshots showing the Agent View Camera in action. The left shows the agent standing on a hill while the right shows the agent looking down into a valley. Notice how in both situations the player has a good view of the agent and its near vicinity.

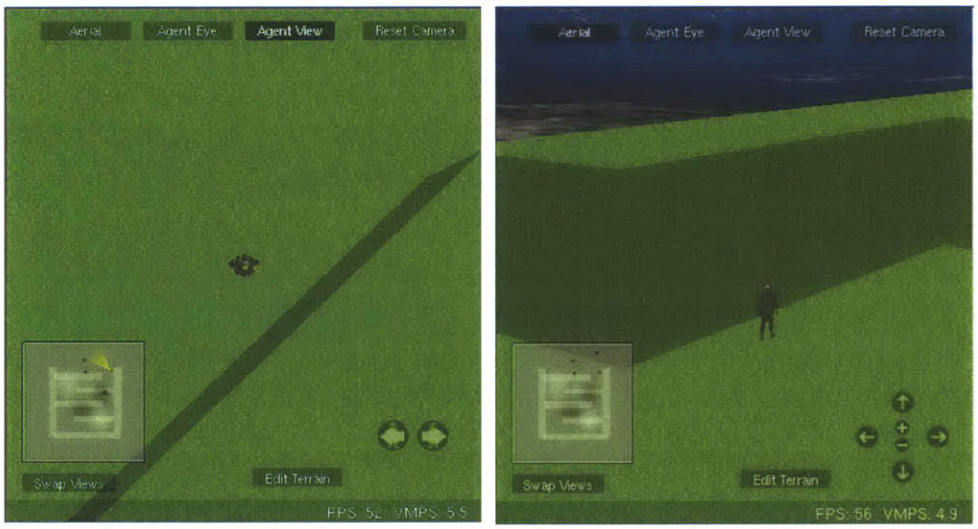


Figure 23: Screenshots showing how the camera will rotate above obscuring terrain. In this case, the agent is right next to a wall. The left shows the overhead screenshot produced by the Agent View camera. The right uses the Aerial camera to more accurately depict the position of the left screenshot.

The combination of all of the calculations in section 6.2 allows the Agent View camera to rotate over terrain that would obscure the viewer. The pseudocode that illustrates how all of these calculations are performed is shown below:

-Find the position, direction, and up vectors of the agent
-Position the camera at the agent's "eye level"
-Calculate the point the camera should look at called the "look at" point, which is globally above the agent

-Translate the camera backwards in the opposite direction of the agent
-Generate the tunnel points (from the camera's position to the agent)
-Check tunnel points for obscuring terrain and find the height the camera needs to be at
-Based on this height, move camera to necessary point along its vertical path to see agent:
    -If camera ends up higher than the maximum altitude, position at top of ellipse
    -If above agent's height but below maximum altitude, position on ellipse
    -If below agent's height, adjust height without changing horizontal distance to agent
-Use a vertical rotation to point camera at camera's "look at" point. This changes the direction as well as the up vector.
-Camera smoothening described in section 8

Section 7: Agent Eye Camera

The 1st person perspective Agent Eye camera also underwent changes, but not as many as the Agent View camera. The main change made to the Agent Eye camera was the removal of the vertical and horizontal tilt to make the camera less disorienting. Even though the Agent Eye camera also had the problem of a small field of vision, we can't reposition this camera since moving it farther back would defeat the purpose of having the camera be from the 1st person perspective. Instead, I increased its field of vision using Camera Blocks, which I will discuss later in section 9.

Section 7.1: Previous Code for Agent Eye

The code for the previous Agent Eye camera was a subset of the code for the previous Agent View camera. The previous Agent Eye camera was positioned at the agent's "eye level" with the camera's up and direction vectors set as the agent's up and direction using the localToGlobal and localToGlobalDirection methods respectively the same way as described in section 6.1. Instead of translating the camera back, the camera intuitively stays at the "eye level" since a 1st person perspective is viewed from the agent's point of view. And during gameplay, this camera also used the camera smoother described in section 8.

Section 7.2: New Code for Agent Eye

The previous code performed as a satisfactory camera, although sometimes the tilt was too disorienting to the gameplay. Having the tilt can create confusing perspectives when the

agent is looking straight down into a valley or straight up at the sky. As mentioned earlier, these perspectives can confuse the user about what he/she is looking at, and give no information about what is happening in the game.

To stabilize the camera and make Starlogo games easier to control, I removed the horizontal and vertical tilt from the camera while leaving the camera position where it was using the previous code. To remove the tilt, I simply set the component of the direction vector to 0 and set the camera's up vector to (0, 1, 0). Pointing the camera in the agent's direction while horizontal to the x-z plane removes the vertical tilt, and pointing the up vector in the y direction removes the horizontal tilt.

One of the resulting screenshots from the new Agent Eye camera is shown in figure 24. The tilt has been removed making the agent's easier to control and easier to see, which is very important for Starlogo games. Even though removing the tilt reduces the view when moving up a hill or down a valley since the user cannot see upwards or downwards, the camera blocks compensate for this by allowing the user to look farther in all directions at his/her own will which will be described more in section 9.

Figure 24: Screenshots showing the new Agent Eye camera. The left is a screenshot using the Agent Eye camera, and the right is a screenshot using the Aerial camera to show where the agent is actually standing. Note that the Agent is actually standing tilted on a hill, but the tilt has been removed in the Agent Eye camera, making it easier to see how the rest of the Starlogo world looks.

Section 8: Camera Smoothness Calculations

In addition to changing the behavior of the cameras, another part of this project was improving the smoothening of the camera motion. There are two types of smoothening implemented. The main smoothening occurs after calculating the camera position and orientation for either the Agent Eye or Agent View camera to gradually move between different camera positions. The other smoothening can be thought of as a "higher order smoothening" used only by the Agent View camera to make sure the camera doesn't move excessively when looking around in the Starlogo world. Both types of smoothening are necessary for good gameplay since without it, motions can be disorienting and frustrating to deal with for the user. However, smoothening effectively can be quite challenging, especially since the time between frames, which we'll refer to as dt (dt = "delta time"), can vary quite a bit depending on the Starlogo project being run.

Section 8.1: Smoothening Used by Both Cameras

Section 8.1.1: Previous Method

The existing system for camera smoothening occurred at the end of updateTurtleCameras() in TorusWorld.java with a one line call to CameraSmoother.smoothCamera shown in figure 25.

```
camSmoother.smoothCamera(camera, deltaTime);
```

Figure 25: Where the camera smoothening occurs.

This method receives two arguments:

- "camera": The instance of the camera we're using. It has the next position, direction, and up vectors that the camera should eventually get to. I will refer to these as the target position, target direction, and target up vectors respectively.

- "deltaTime": This is the amount of time since the previous frame.

The main purpose of smoothCamera() is to find the next position and orientation of the camera given the target position and orientation, the time since the last frame, and a couple of the previous camera positions and orientations which the CameraSmoother class keeps in memory. The target position and orientation are the resulting camera position and orientation from the methods for Agent View or Agent Eye described in sections 6 and 7 respectively. In order to create smooth movements, the CameraSmoother simulates a spring force between the

camera's current position and its target position. A similar type of motion is performed for the camera's orientation. This process allows the camera to take multiple frames to get to where it should be by smoothly accelerating and decelerating to its target position and orientation. Figure 25 illustrates an example of these calculations.

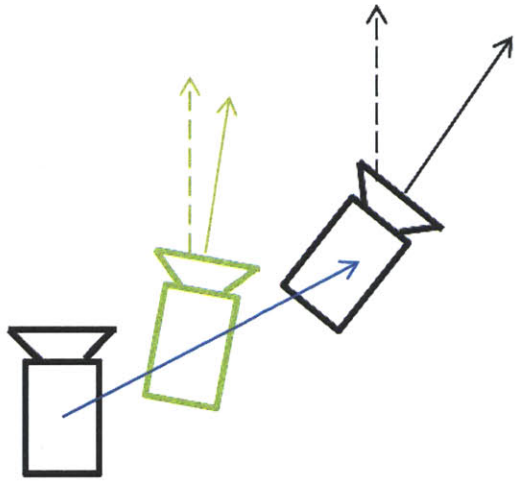


Figure 25: Figure showing how camera smoothing works. The blue arrow points from the camera of the current frame to the target position and orientation based on player input. The green camera represents the position that the camera smoother may output on the first time step. Notice how it takes less movement to get to the green camera from the current position. This allows the camera to make smoother motions. The dashed arrows are simply showing the vertical direction (that the first camera is in) and the other arrows simply show the direction of the camera.

In order to calculate the next position of the camera, the CameraSmoother class uses "backwards" or "implicit" Verlet Integration. Standard Verlet Integration is a process that over time calculates the trajectory of an object given the forces acting on it. The formula itself approximates the next position of an object given the forces acting on it at that point in time, a couple of previous positions, and the amount of time between positions. Visit this reference: http://en.wikipedia.org/wiki/Verlet_integration for a derivation of the formula. Implicit Verlet Integration is a more stable form of regular Verlet Integration by using a slightly different equation; it's analogous to how the Implicit Euler method is a similar form of the regular Euler method. The CameraSmoother uses Implicit Verlet Integration to calculate the trajectory of our camera.

In the context of Starlogo, for the camera position we simulate a spring between the camera's current position and its target position. This calculation is performed at each time step so as the camera gets closer and closer to its target position the spring force is less and less. It's necessary to recalculate the spring force each time since the agent could be moved before the camera reaches its target. In addition to a spring force, we also use a damping force to reduce oscillations in the camera's movement and make its movement smooth.

The code that uses a modified version of Implicit Verlet Integration used to calculate camera position is shown in figure 26. The equation modifies Verlet slightly by using dtr, the ratio of the current dt and the previous dt where the regular Verlet Integration treats the dts as equal and thus treats this ratio as 1. There was no documentation on the choice of the spring and damping constants, but I presume these were chosen by experimentation to make Starlogo camera motion smooth and responsive.

```
public static double K = 50;// 140;//20;//50; //Spring Constant
public static double F = 12;// 1; //15; //Damping Constant

private class VectorSmoother {
	double ux, uy, uz;
	double vx, vy, vz;

	int step = 0;
	public void reset() {
		step = 0;
	}

	public void smoothVector(Vector3f target, double dt, double dtr) {
		double tx = target.x, ty = target.y, tz = target.z;
		step++;
		if (step > 2) // if we have enough data
		{
			double k = K * dt * dt;
			double f = F * dt;

			tx = 1.0 / (1.0 + k)
				* ((1 + dtr - f) * ux - (dtr - f) * vx + k * tx);
			ty = 1.0 / (1.0 + k)
				* ((1 + dtr - f) * uy - (dtr - f) * vy + k * ty);
			tz = 1.0 / (1.0 + k)
				* ((1 + dtr - f) * uz - (dtr - f) * vz + k * tz);

		}

		//Storing data for next timestep
		vx = ux;
		vy = uy;
		vz = uz;

		ux = tx;
		uy = ty;
		uz = tz;

		target.set((float) tx, (float) ty, (float) tz);
	}
}
```

Figure 26: Figure showing the main segment of code for using Implicit Verlet Integration to calculate the camera's position.

Smoothing of the camera orientation is handled in a very similar fashion. The orientation of the camera can be represented as a 3x3 rotation matrix to transform global

coordinates into the camera coordinate system. The methods computeRotationMatrix() and fromRotationMatrix() in the Camera class calculate the 3x3 matrix from the direction and up vectors and vice-versa respectively. A 3x3 rotation matrix can also be represented as a quaternion (Waveren 2-3), which provides an easy way to calculate linear interpolation (lerp) and spherical linear interpolation (slerp). Although the calculation using the quaternions isn't identical to the position calculations, it's very similar to Implicit Verlet Integration except we need to use lerp and slerp instead of simple addition. The snippets of code that performs these calculations is shown in figure 27.

```
private class MatrixSmoother {
	private Quaternion x1 = new Quaternion();
	private Quaternion x2 = new Quaternion();

	int step = 0;

	public void reset() {
		step = 0;
	}

	public void smoothMatrix(Matrix3f target, double dt, double dtr) {
		step++;

		Quaternion xt = new Quaternion();
		xt.fromRotationMatrix(target);

		if (step > 2) // if we have enough data
		{
			double k = directionKScale * K * dt * dt;
			double f = F * dt;
			xt = Quaternion.slerp(
				Quaternion.lerp(x2, x1, 2 * (1 + dtr - f) / (1 + k)),
				Quaternion.lerp(x2, xt, 2 * k / (1 + k)),
				0.5);
			target.set(xt);
		}
		x2 = x1; // x2 is the 2nd previous rotation quaternion
		x1 = xt; // x1 is the 1st previous rotation quaternion
	}
}
```

Figure 27: Figure showing the main segment of code for using a method similar to Implicit Verlet Integration to calculate the camera's orientation. Note the calculations are nearly identical, except we use lerp and slerp for quaternions.

One last part of the existing code that handles the camera smoothening is the reset() method. This method immediately sets the current camera position and orientation equal to the target position and orientation, and clears the data for the previous positions and orientations. It's used to move the camera directly to its target position whenever the dt is too big. When the dt is too big, the Verlet method becomes unstable, meaning the positions and

orientations returned by our methods oscillate and diverge. This was the main problem I faced which I will discuss next.

Section 8.1.2: Problem with Verlet

The main issue that I came across with the existing smoothening system is that for large values of dt, the calculations become unstable. This is illustrated more in figure 28 where the camera can overshoot the target since we're multiplying the spring force by too large of a time. If the dt rises past the point where it starts overshooting, the camera oscillates when it's trying to get to its target; the camera first overshoots a little, then the spring force draws the camera back towards the target and so on. This type of motion can be very disruptive to gameplay. Although figure 28 only shows how this can happen with the position, this can also happen with the direction as well. These oscillations become even larger with larger dt. At a certain point, the oscillations grow out of control and the camera starts moving without bound making games completely unplayable. I will refer to this point as the dt that causes the camera to become unstable. This is shown more in figure 29.

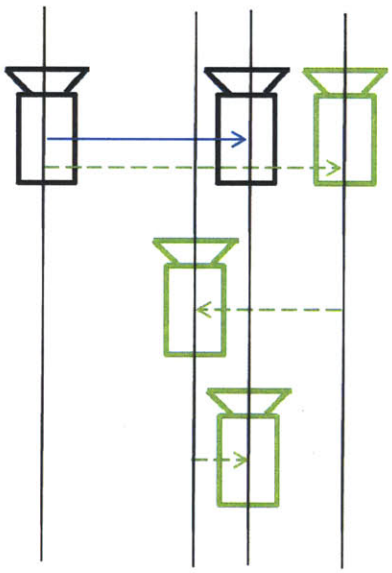


Figure 28: Figure showing oscillation behavior of the camera when dt is large enough. The dt chosen here makes the camera overshoot the target by a little, but the camera motion will eventually get to the target. The black cameras show the initial position and the target position, and the green cameras show the calculated positions by the smoother.

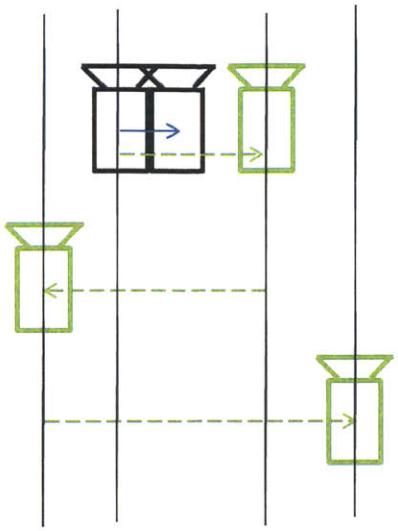


Figure 29: Figure showing exploding behavior of the camera when dt is chosen large enough. The camera goes from the initial position to a position that is farther away from the target. This keeps happening with each frame to make the camera's position oscillate out of control.

To account for the potential instability of Verlet Integration, CameraSmoother has a reset() method described earlier. While this does allow our camera system to automatically stabilize when the dt gets very high, this can make the camera motion look very glitchy. This problem can be very difficult to deal with when there are a lot of calculations to compute. Large amounts of calculation causes the dt to be high on certain frames, which causes the camera to lurch forward every so often, which can be very disruptive to gameplay. Large values of dt are also caused by the user focusing away from the window. Starlogo allocates a lot of CPU when the user is focusing on the Spaceland window and/or is running a Starlogo project but doesn't allocate a lot if neither is true. However, the camera might still be moving a short while after the user stops running a project and focuses away from the window. This camera movement would also have high values of dt. The glitchy motions of the reset() function was the main problem of camera smoothing that I spent a lot of time trying to fix in order to improve gameplay for the user.

Section 8.2: Choice of constants

In order to fix the glitchy motions of the reset(), I removed the code where reset() was called if the dt was too high. This eliminated the jerky motions of the camera but brought back the problem of unstable camera motions for high values of dt. To stabilize the camera motions, I experimented with changing the constants for the spring and the damping force.

I found that I could simulate high values of dt by using Thread.sleep() during the camera calculations since this would increase the amount of time to calculate each frame. I could thus fully explore how manipulating the spring and damping constants would affect the camera smoothening for various values of dt. In both of the position and orientation calculations, both of the constants are multiplied by either dt or dt^2 before they are added into the equation. After some experimentation, I found that raising the spring constant or decreasing the damping force constant will help stabilize the camera motion. This held true for any value of dt that I tried including values of dt as high as 0.3, which is well beyond what the dt should be when using Starlogo.

To get a general sense of why this occurs, it's clear that using very high values of k will simply set the camera to its target position and orientation. Using smaller values for the damping constant won't necessarily reduce camera motion, although it does prevent the contributions from the damping force from getting too high. Both calculations for the position and orientation include the term dt* (damping constant) so reducing this value as dt gets higher will help stabilize the camera.

After experimenting with the constants and values of dt, I found that using the previous values K = 50 and F = 15 yielded smooth motion for low values of dt. Around dt = 0.15, these values caused the camera calculations to become unstable. For higher values of dt, I found that the values K = 130 and F = 7 kept the camera motions stable for the values of dt that I tested while still moving the camera slowly enough to allow the user to get a sense of the camera motion. It should be noted that the camera motions when dt is high aren't nearly as smooth as those when dt is low but this happens with any choice of the constants since the same motions are shown with fewer frames for high dt. The alternate choice of constants primarily allows the camera motions to be stable without immediately positioning the camera at the target position and orientation like the reset() method did.

To switch between the constants, I chose dt = 0.1 as a cutoff to ensure that the camera motions are stable. However, instead of using a simple cutoff, I keep track of the average dt for the last 5 frames. An average dt that is greater than 0.13 would trigger using the constants for higher dt, and an average dt that is lower than 0.07 would trigger using the constants for lower dt. The reason for using average dt instead of dt for individual frames is that dt can vary a bit especially when Starlogo games are computationally intensive. Switching between sets of constants disrupts camera motion so we want to switch constants only when necessary. Using the average dt reduces variation and requiring that the average to be slightly beyond the cutoff will help ensure that the constants are switched only when the dt has substantially changed. This new system of using constants allows the camera motions to be stable for reasonable values of dt in addition to showing more motion than simply setting the camera to its new

position and orientation. I'll now briefly explain the additional smoothening I used for the Agent View camera.

Section 8.3 Agent View Additional Smoothening

I want to briefly mention the additional form of smoothening that I added for the Agent View camera. The smoothening that I added occurs after we calculate the translation of the camera height and before we calculate the new camera position and orientation. The code for the translation along with pseudocode for the surrounding calculations for Agent View is displayed in figure 30.

```
//Pseudocode Before:
-Check tunnel for obscuring terrain, find needed translation upwards, store this in variable
translationDistance

float nextTranslation = prevTranslation + scaleTranslation *
 (translationDistance - prevTranslation);

float nextY = nextTranslation + pos.y; //The y coordinate we'll end up at


//Pseudocode After:
-Move camera to point along path at height nextY
-Main form of camera smoothening
```

Figure 30: Figure showing the code and pseudocode for how the extra Agent View smoothening is implemented.

The extra smoothening effectively makes the movement along the camera's vertical path described in section 6 take a few frames to take effect instead of happening all at once. It's helpful to specifically slow down movement due to obscuring terrain since when moving the camera around, the amount of this movement can change much faster than other movements of the camera. If the agent is positioned very close to a wall, the camera needs to move to a very high height and as a result may rotate overhead in order to see the agent. As the camera rotates around, the camera will move upwards along its path when the camera is behind the wall and as soon as it moves away from the wall, the camera will move back downwards along its vertical path. This can be very disorienting if the camera keeps going back and forth behind a wall. To reduce this disorienting motion, the extra smoothening will slow down these motions which prevents the camera from translating too far if it only needs to translate temporarily.

Together, the two types of smoothening help to keep the camera motions smooth during Starlogo gameplay for all reasonable values of how much time elapses between rendered frames. The last section of the project involves giving the user greater control of the camera during Starlogo games and simulations which I will discuss next.

Section 9: Camera Blocks

The last part of this project was the Camera Blocks, puzzle blocks that students can use when programming their games to manually control the camera. This allows the user to move the camera to the view that the user desires, including the requested second-person perspective and the useful agent overhead perspective. The modifications that the blocks make are in addition to the previous camera functionality; for instance if the blocks rotate the Agent View camera to a new position, the same terrain checking will occur and the camera will rotate over obscuring terrain from its current position. In this sense, the user doesn't have absolute complete control over the camera, but these blocks allow the user a much wider range of control and allow the user to be much more immersed into Starlogo games.

Section 9.1: Previous Code

Previous to this project, blocks that could manually control the position and orientation of the camera didn't exist at all. The only blocks relevant to the camera were blocks that chose which camera is currently being used as well as a few others. The blocks that tell the user which camera is being used and set which camera is being used are shown in figure 31 as the oval blocks and rectangular blocks respectively. Note that the blocks relating to the Agent View camera are labeled as "over shoulder?" and over shoulder to more accurately describe where the camera is. The two blocks shown in figure 32 tell the user which agent the camera is following and set which agent the camera should be following. These are only really used when using the Agent Eye and Agent View cameras are being used. These will not have much effect when using the Aerial camera.

Figure 31: Blocks used to tell the user which camera is being used, and set which camera is being used.

Figure 32: Blocks used to tell the user which agent the camera is following and set which agent the camera should be following.

The other blocks that existed previously that were relevant to camera controls are shown in figure 33. The three blocks control the "overhead" view. The overhead view switches the main view of the terrain and the map view shown in the lower left corner. The map view camera has a fixed position above the terrain and looks straight down at the terrain. This perspective is unlike all of the other cameras since this camera is immutable. Note that even though it does provide an overhead view as we wanted, the camera is still positioned very far away from the terrain and doesn't follow individual agents. This makes this perspective difficult to use in a game since the user might have trouble seeing what is happening in the terrain similar to the aerial camera as mentioned in section 5.

The blocks that tell the user whether it is in use and turn it on and off are shown in the top row of figure 33 from left to right respectively. A screenshot showing overhead view is shown in figure 34.

Figure 33: Other previously existing blocks relevant to camera controls.

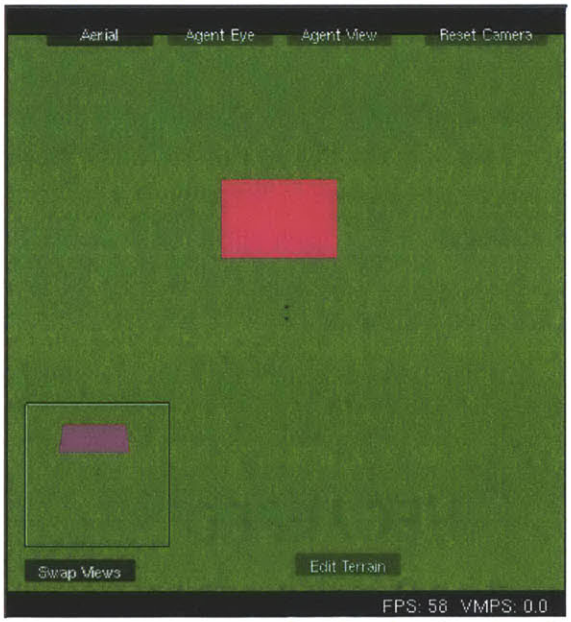


Figure 34: Screenshot showing the overhead view that can be triggered by the camera blocks in figure 33. Note how the magenta rectangle that is painted on the terrain is viewed straight on from the main perspective and from an angle in the map view in the lower left corner.

These are all of the previously existing blocks that manipulated the camera perspective. In the next section I will describe the function and implementation of the new camera blocks.

Section 9.2 New Camera Blocks

The new blocks I created allow for much greater control of the Agent View, Agent Eye, and Aerial cameras. The blocks I created are shown in figure 35. These blocks allow the user to shift the perspective in a variety of directions and have greater control of the camera. They also allow for a much larger field of vision and allow the user to look around the terrain with much greater flexibility. The behaviors of these blocks vary depending on which camera is currently being used which will be explained shortly. It should also be noted that the effects of the camera blocks are separate for each camera however so altering the perspective of one camera will not affect a different camera. By allowing greater control of the camera, the user will become more immersed in his or her Starlogo world, which greatly improves the gameplay of Starlogo games and simulations.

Figure 35: Screenshot showing the new camera blocks to allow the user to have greater control over the game perspective.

Section 9.2.1 Agent Eye Camera Controls

The controls for the Agent Eye camera allow the user to look around the Starlogo terrain from an enhanced first person perspective. The four blocks on the left column of figure 35 are "look left", "look right", "look up", and "look down" which will rotate the camera the specified number of degrees to allow the user to look in the respective direction. To implement these in the code, I created the methods lookRight and lookup in TorusWorld.java that are called whenever the blocks are executed. The blocks "look left" and "look down" also use the methods lookRight and lookUp, except they call the respective methods with a negative rotation. The methods are shown in figure 36. Each time the blocks are executed, static variables that keep track of the cumulative horizontal and vertical rotations are updated. Note that there are separate variables for each of the camera movements for each of the three cameras in order to keep the camera modifications separate. Thus when using the Agent Eye camera using rotations will only affect the variables for Agent Eye camera rotations as shown in

the code in figure 36. Also, for the Agent Eye camera specifically, the rotation variables are clipped at certain values to prevent the camera from getting into a bad state. For Agent Eye, the vertical rotation is clipped to be between -75 and 75 degrees since the Starlogo camera is not currently configured to rotate upside down. Horizontal rotation is not clipped at all since the camera is free to look horizontally in any direction.

```
    //Degrees can be negative if we're looking to the left.
    public static void lookRight(float degrees) {
      if (SLCameras.currentCamera == SLCameras.PERSPECTIVE_CAMERA) {
            horizontalRotationAerial += degrees;
      }
      if (SLCameras.currentCamera == SLCameras.OVER_THE_SHOULDER_CAMERA) {
            horizontalRotationShoulder += degrees;
      }
      if (SLCameras.currentCamera == SLCameras.TURTLE_EYE_CAMERA) {
            horizontalRotationEye += degrees;
      }
    }

    //Degrees can be negative if we're looking down instead.
    public static void lookUp(float degrees) {
      if (SLCameras.currentCamera == SLCameras.PERSPECTIVE_CAMERA) {
            verticalRotationAerial += degrees;
      }
      if (SLCameras.currentCamera == SLCameras.OVER_THE_SHOULDER_CAMERA) {
            verticalRotationShoulder += degrees;

            verticalRotationShoulder = Math.max(verticalRotationShoulder, -
85.0f);
            verticalRotationShoulder = Math.min(verticalRotationShoulder,
20.0f);
      }
      if (SLCameras.currentCamera == SLCameras.TURTLE_EYE_CAMERA) {
            verticalRotationEye += degrees;

            verticalRotationEye = Math.max(verticalRotationEye, -75.0f);
            verticalRotationEye = Math.min(verticalRotationEye, 75.0f);
      }
  }
```

Figure 36: Figure showing the code for the lookRight and lookup methods.

After the rotation variables are updated, these rotations are used in TorusWorld.updateTurtleCameras(), the method described in sections 6 and 7 that updates the Agent View and Agent Eye cameras. The horizontal and vertical rotations are performed about the current camera position and occur after the rest of the code for Agent Eye that was described in section 7. The results of horizontal and vertical rotations are shown in figure 37 and 38. Figure 37 shows the unchanged view on the left, and then the view modified by "look left" and "look right" in the center and right respectively. Figure 38 shows the unchanged view on the left, and then the view modified by "look up" and "look down".

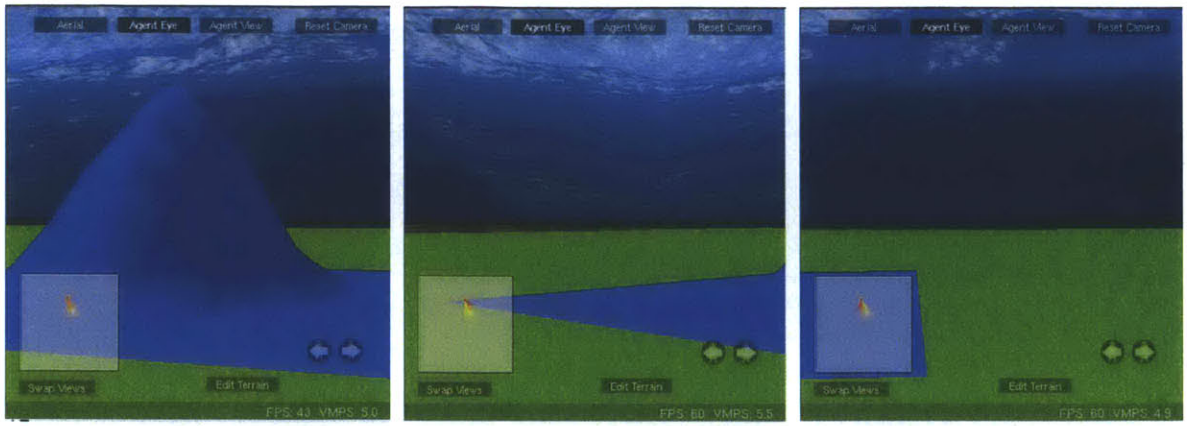


Figure 37: Figure showing the effects of look left and look right. The left image is the unchanged image, the center image is looking to the left, and the right image is looking to the right.

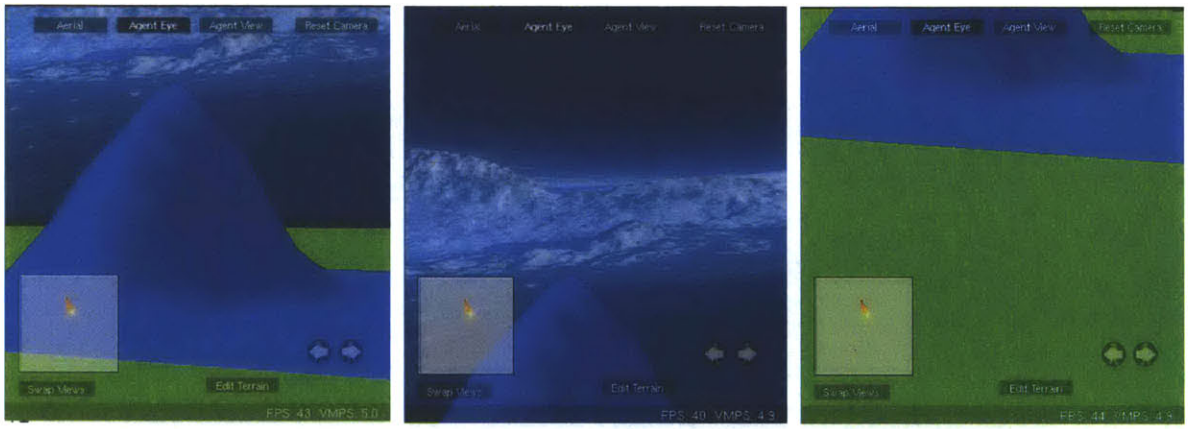


Figure 38: Figure showing the effects of look up and look down. The left image is the unchanged image, the center image is looking up, and the right image is looking down.

Two of the other camera blocks are "look zoom in" and "look zoom out". These two blocks allow the user to zoom the camera in to view things that are farther away. Similar to the other rotation blocks, I created a zoomIn method shown in figure 39 that is called when these two blocks are executed in order to update the cumulative zoom variable. Also similar to the other blocks, there are separate zoom variables for each of the cameras. When updating, the zoom variable cannot go below zero since when using a 1st person perspective it's impossible to place the view behind the agent. The "look zoom out" block also uses the zoomIn method except with negative distance values. Zooming occurs after rotations in Agent Eye by translating the camera forwards or backwards in the direction the camera is facing. An example of zooming in is shown in figure 40 where the zoom is applied to the image on the right.

```
//Distance can be negative if we're zooming out instead.
    public static void zoomIn(float distance) {
      if (SLCameras.currentCamera == SLCameras.PERSPECTIVE_CAMERA) {
```

```
            zoomAmountAerial += distance;
        }
        if (SLCameras.currentCamera == SLCameras.OVER_THE_SHOULDER_CAMERA) {
            zoomAmountShoulder += distance;

            // This value should never go above cameraDistance - 10 and should
never go below -10
            zoomAmountShoulder = Math.min(zoomAmountShoulder, cameraDistance
- 10);
            zoomAmountShoulder = Math.max(zoomAmountShoulder, -10);
        }
        if (SLCameras.currentCamera == SLCameras.TURTLE_EYE_CAMERA) {
            zoomAmountEye += distance;

            // Make sure that zoomAmountEye isn't below zero.
            zoomAmountEye = Math.max(zoomAmountEye, 0.0f);
        }
    }
```

Figure 39: Figure showing the code used for the zoomIn method in TorusWorld.java.

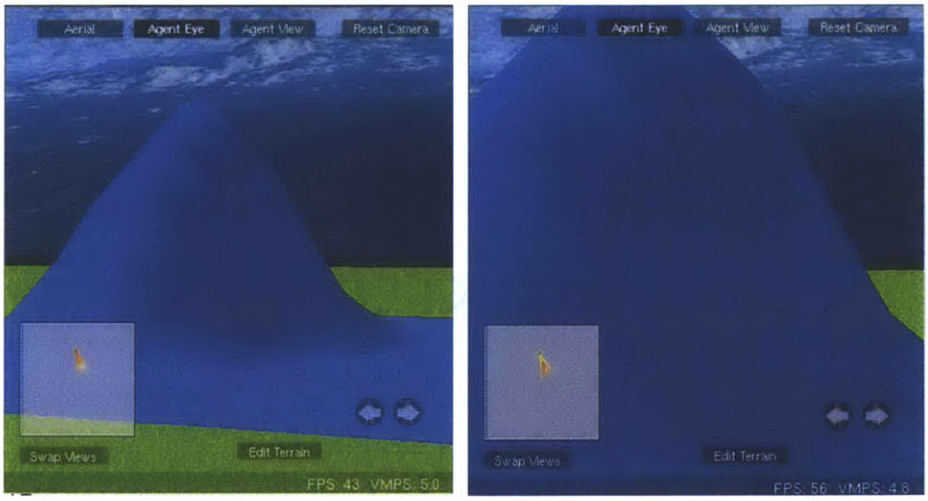


Figure 40: Figure showing the effects of "look zoom in". The unchanged perspective is shown on the left, and the zoomed in shot is on the right.

Also note that in each of the screenshots, there is a red cone in addition to a yellow cone in the mini-map which is in the lower left corner of the window. The yellow cone existed previous to this project and shows the field of vision of the controlled agent with respect to the rest of the terrain. I implemented the red cone during this project to show the field of vision of the camera. This is helpful since when using zoom and rotations to modify the Agent Eye camera, there are no other indications to the position and orientation of the camera relative to the position and orientation of the agent.

The last camera block is the "look reset" block. Execution of this block removes the modifications caused by rotations and zooming for the current camera. In the code this simply means setting the variables that keep track of the cumulative rotations and zooming equal to 0. This block works the same for each of the three cameras.

The camera blocks greatly improve the use of the Agent Eye camera by giving the user more control over the direction and position of the camera. The user has a much greater field of vision, which greatly increases the gameplay of Starlogo games. The usefulness of the camera blocks is also extended to the other cameras such as Agent View, which I will talk about next.

Section 9.2.2 Agent View Camera Controls Description

The controls for the Agent View camera allow the user to look around the Starlogo terrain while keeping the agent in his/her perspective. Similar to how the camera blocks worked for the Agent Eye camera, using the camera blocks for the Agent View camera uses the same methods lookRight, lookUp, and zoomIn described in section 9.2.1 and we perform the additional rotations and zooming within the code for the Agent View camera described in section 6. The functionality for the Agent View camera controls is a little different due to the nature of the 3rd person perspective and the camera modifications happen in a different order in the code. The code changes necessary for the rotations and zooming are shown in figure 41, as well as pseudocode for the calculations that come before and after it which were described in section 6. I will now explain the code changes for the Agent View camera controls.

```
//Pseudocode before:
-Find the position, direction, and up vectors of the agent
-Calculate the point the camera should look at, which is globally above the agent


//Code changes
// This is the angle between the direction and the horizontal
double angle = Math.asin(Math.min(Math.abs(dir.y), 0.999));

// Use the angle to translate backwards, also taking the zoom into account
float translateBackwards = (cameraDistance - zoomAmountShoulder)/
        (float)Math.cos(angle);

camera.translate(0, 0, translateBackwards); // Translate the camera backwards
camera.translateGlobal(0, -1.0f*verticalRotationShoulder, 0);

Vector3f newDir = Vector3f.subtract(posLookAt, camera.getPositionCopy());
newDir.normalize();
```

```
//Find the angle of rotation between the two
float dotProd = Vector3f.dot(dir, newDir);
dotProd = dotProd / dir.length() / newDir.length(); //Both should already be
normalized, but just to make sure.
double ddotProd = (double)dotProd;
ddotProd = Math.min(0.999, ddotProd); //To ensure that dot product is less
than 1 (so cosine of it won't return NaN
float angleOfRotation = (float)Math.acos((double)ddotProd);

if (verticalRotationShoulder <= 0) {
        camera.rotateVerticallyAround(camera.getPositionCopy(), -
1*angleOfRotation);
}
else {
        camera.rotateVerticallyAround(camera.getPositionCopy(),
angleOfRotation);
}

//Rotations due to look left and look right and zoom in.
camera.rotateHorizontallyAround(posLookAt,
        horizontalRotationShoulder*(float)Math.PI/180.0f);


//Pseudocode after
-Generate the tunnel points (from the camera's position to the agent)
-Check tunnel for obscuring terrain, find needed translation upwards
-Move camera to necessary point along path to see agent, using higher-order smoothening
described in section 8
-Regular camera smoothening described in section 8
```

Figure 41: Pseudocode and code describing how the camera controls for Agent View are implemented with respect to the rest of the code for Agent View.

As seen in the code, the zooming calculations happen first. The Agent View zoom variable zoomAmountShoulder modifies the horizontal distance the camera will translate backwards, which modifies the actual distance the camera translates backwards. This is demonstrated in figure 42, where the camera in black represents the unchanged camera, the blue camera has been zoomed in, and the green camera has been zoomed out. The zooming variable for Agent View is constrained between -10 and cameraDistance - 10 as shown in figure 39. This simply constrains the zoom so the user doesn't zoom in past the agent and doesn't zoom out far enough to the point where the camera becomes difficult to use.

By changing the amount the camera translates backwards, this effectively changes the starting point of the camera. The tunnel of points that is checked for obscuring terrain will become longer or shorter depending on the zoom which allows the camera to still rotate above obscuring terrain whether or not its positioned closer or farther away. The other change needed for Agent View camera zooming is scaling the camera path that was shown in figure 19.

In order to keep zooming consistent for all positions on the path, the path needs to be scaled evenly to ensure that zooming always brings the camera closer or farther from the agent. This scaling is demonstrated in figure 43. In order to change this, I only needed to change the minor and major axes of the ellipse which were namely the variables initialHorizontalDistance and maxAltitude in the equation shown in figure 20. I set initialHorizontalDistance equal to the new horizontal distance caused by the zoom which is namely cameraDistance – zoomAmountShoulder. I set maxAltitude equal to twice this amount to preserve its value of 60 when the horizontal distance is 30 in addition to allowing the ellipse to scale. Screenshots showing zooming in and out are shown in figure 44 in the center and on the right respectively with the unchanged image on the left.

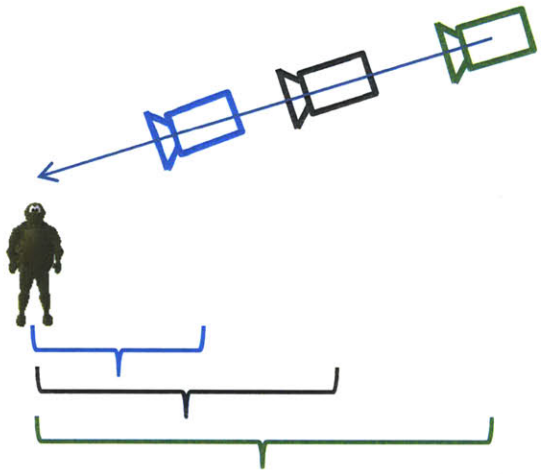


Figure 42: Effect of zooming the Agent View camera in and out. The black camera is the unchanged camera, which is at a horizontal distance of 30. The blue camera is zoomed in and has a shorter horizontal distance, while the green camera is farther and has a longer horizontal distance.

The calculations for the "look up" and "look down" blocks occur next in the code. Vertical rotations for the Agent View camera allow the user to look up and look down from the current perspective but work differently than vertical rotations for the Agent Eye camera. Since the vertical path of the camera shown in figure 19 is not a uniform shape but instead composed of multiple parts, vertical control is much trickier to think about. It's also trickier since I want the camera to automatically rotate above obscuring terrain even if the vertical rotation would position the camera otherwise.

To keep things simple, I used the input instead as modifications to its vertical height, which will move the camera along its vertical path. For instance, a "look down" command will move the camera upwards along the path to allow the user to look downwards. The camera is also rotated vertically so the camera will look at the "look at" point described in section 6. By simply moving along the path we already created, this ensures smooth camera movement. In

addition, by placing it before the terrain checking, we also ensure that the camera will still follow the previously defined rules to rotate above obscuring terrain.

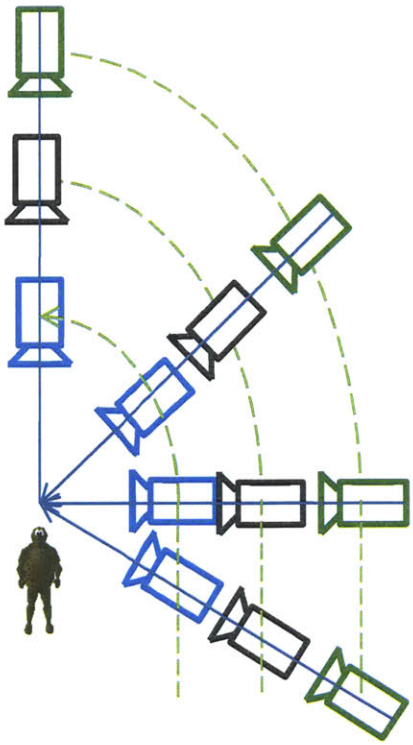


Figure 43: Figure showing how the path of the camera scales when the camera zooms in and out. The blue cameras show the zoomed in path and the green cameras show the zoomed out path. Note that by scaling the path, the camera always moves straight outwards which provides intuitive camera movement.

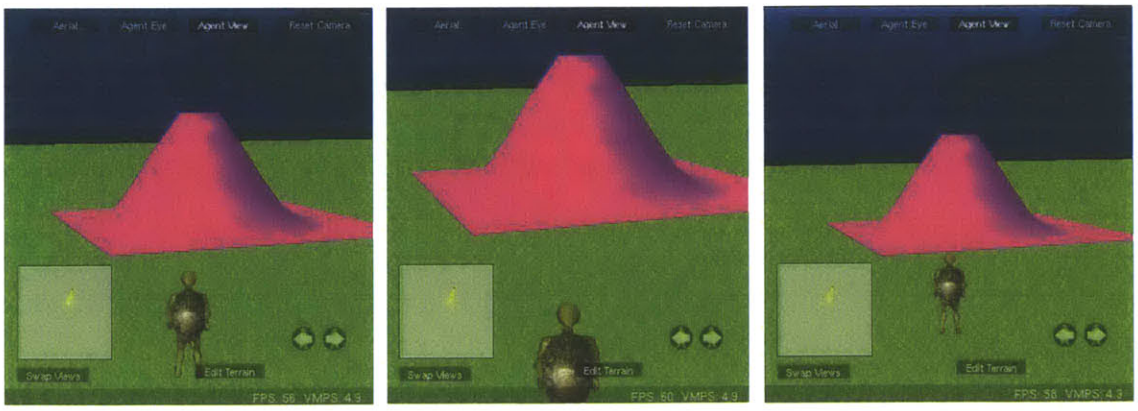


Figure 44: Screenshots illustrating zooming the Agent View camera. The left image is the unchanged view, the center image is a zoomed in view, and the right image is a zoomed out view.

Admittedly this isn't the perfect solution since the user could potentially get into situations where executing "look up" wouldn't look up even if the camera could. For instance, if the vertical rotation variable would translate the camera above the top of the camera path, a "look up" command, which would decrease camera height, might leave the camera above the top of the path. In this case the user would have to continue executing "look up" commands in order to decrease the camera height and allow the user to look upwards. Despite the weakness however, this camera action is consistent with the others since it is a cumulative action that allows the user to look up or down from the current perspective. In addition, to reduce the problem I constrained the vertical camera movement to be 85 at most upwards and 20 at most downwards. This allows the camera to rotate all of the way to the overhead view most of the time in addition to slightly below horizontal while not allowing this variable to get too large in magnitude. Example screenshots of looking upwards and downwards are shown in figure 45, where the left most image is the regular Agent View perspective and the right two images are of the perspective looking upwards and downwards respectively. It should also be noted that "look down" allows the user to use an overhead view that tracks the agent as mentioned earlier. Instead of using a separate camera, we can seamlessly integrate this view in with the Agent View camera. A screenshot of this overhead perspective is shown in figure 46.

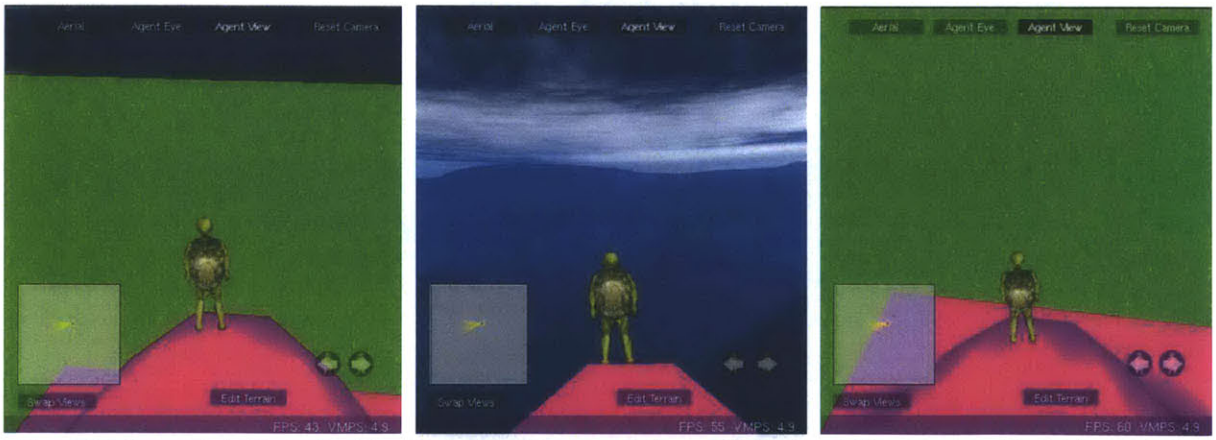


Figure 45: Screenshots illustrating the effects of looking up and down with the Agent View camera. The left image is the unchanged image, the center is looking up, and the right is looking down.

The "look left" and "look right" blocks rotate the camera in a horizontal circle around the agent to allow the user to look to the agent's left and right respectively. In the code, the camera is rotated horizontally around the "look at" point to change the starting position of the camera. The code then checks for obscuring terrain from the new position and rotates the camera above the terrain as necessary. Example screenshots of looking to the left and right are shown in figure 47 where the left most image is the regular Agent View perspective and the right two images are of the perspective looking left and right respectively. It should also be

noted that a horizontal rotation of 180 degrees allows the user to achieve the requested second person perspective. In a similar fashion to the overhead view that tracks the agent, this view is integrated seamlessly into the Agent View camera which eliminates the need for an additional camera and allows the user greater control of the perspective. A screenshot of second person perspective is shown in figure 47.

Figure 45: Screenshot showing the overhead perspective using the "look down" functionality of the Agent View camera.

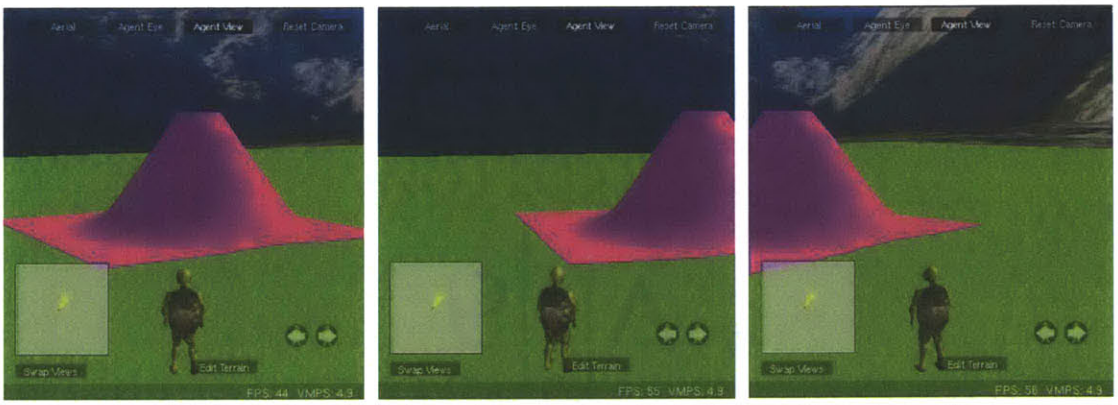


Figure 46: Screenshots showing horizontal rotations using Agent View. The left image is the unchanged image, the center uses "look left" and the right uses "look right"

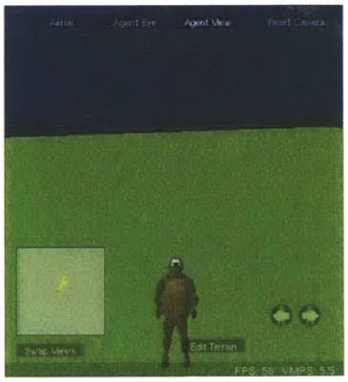


Figure 47: Screenshot showing the second person perspective that can be attained using horizontal rotations using the Agent View camera.

9.2.3 Aerial Camera Controls

The Aerial Camera controls were added mainly to keep consistency in having the camera blocks allow the user to look and zoom in different directions. Since the Aerial camera doesn't follow an agent, "look up", "look left", "look right", and "look down" simply translate the view upwards, to the left, to the right, and downwards respectively. Zooming works similarly as it translates the camera forwards and backwards to zoom in on the terrain and zoom outwards respectively. This can all be done with the translate function of the camera mentioned earlier. It should also be noted that these camera movements are exactly identical to the movements provided by the buttons in the lower right corner of the terrain window. By creating this functionality in the form of Starlogo blocks, we not only keep the blocks consistent for all of the cameras, but we also allow the user to incorporate the functionality of these buttons into the program which allows for greater control of the view.

The one difficulty that I ran into was that unlike the Agent Eye and Agent View cameras, the Aerial camera doesn't get reset to the agent's position and orientation so after a translation is made, the Aerial camera needs some way of remembering that it already made that translation. To account for this, I implemented variables that kept track of the previous translation. I would then only move the camera by whatever new translation was made. This allows the translations as well as the reset block to function properly.

Section 10: Future Work

This project greatly improved the Starlogo cameras by improving their motion, smoothening, and control. That being said, there is still a lot of potential for future work on the cameras and for Starlogo in general. For the Agent View camera, more work can be done with the vertical camera rotations. The current implementation stays consistent with the other blocks, but ideally there could be a system with uniform vertical motion in addition to eliminating situations where executing the look blocks results in no apparent response. There was not enough time during this project to find this solution, but perhaps with a slightly more complex system for vertical rotations, these extra features could be satisfied. For the Aerial Camera, the camera block functionality could be extended to use rotations instead of simple translations. Depending on the implementation, this could allow the user almost full control over the position and orientation of the camera. More can be done with the camera smoothening as well since although I found constants that appeared to work well, it would be better if there could be more rigorous reasoning. During this project, I attempted to use

system analysis and examine how the constants affect the behavior of the camera motion, although this quickly became difficult since the direction calculations involved slerp and lerp which are non-linear functions. Perhaps with more advanced analysis, more can be explored regarding the smoothening.

As far as future work for Starlogo in general, one possible area to explore is a set of more advanced drawing tools. When designing terrains, some patterns might be needed repeatedly that would involve the user drawing multiple polygons on the terrain. For example, in order to draw the road shown in figure 48, the user needs to draw 7 different polygons. If the user wanted to draw multiple roads, this would require a lot of effort from the user. Perhaps the development of a terrain design copy/paste tool would help the user create more elaborate Starlogo worlds more efficiently. The same sort of idea could apply for terrain features such as walls created for a maze. As shown in figure 49, creating a maze could be very laborious for the user, since the current terrain tools only allow the user to create one wall at a time. It would be more helpful if the user had a tool to create multiple walls at the same time. There still remains a lot of potential for further development of Starlogo to give users an even more exciting experience.

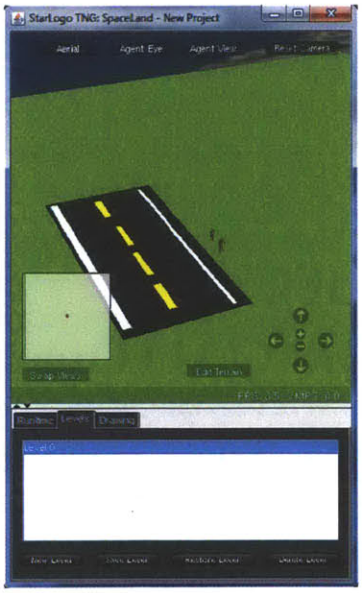


Figure 48: Screenshot showing how a road might be drawn on the terrain. Notice how drawing one such road requires 7 different rectangles drawn by the user.

Figure 49: Screenshot showing how a user might go about constructing a maze. Notice how the user needs to separately select each wall to raise or lower to construct the maze.

Section 11: Conclusion

During this project, I further developed the Starlogo cameras to greatly improve Starlogo gameplay. The Agent View camera now will automatically rotate above obscuring terrain so the user always has an idea of what is happening with their agent in the game. The Agent Eye camera now has no horizontal tilt to give the user a first person perspective without disorienting the user. The smoothening of the camera motions has been improved for a wider range of time taken between frames to prevent jerkiness in camera motion during gameplay. And the creation of camera blocks allows the user to take direct control of the cameras and obtain the view of the user's choice. Together, these improvements will hopefully further improve the gameplay of Starlogo games and enhance the user's experience by being implemented in Starlogo TNG 1.6.

Section 12: Acknowledgements

I would like to extend special thanks to Professor Eric Klopfer and Daniel Wendel for their assistance, guidance, and support throughout this project. I would also like to thank the MIT Teacher Education Program as well as the MIT EECS department for making this thesis project possible. Lastly, I am thankful for all of my family and friends for their support during the completion of this project.

Work Cited:

Burroughs, Mark. "Terrain Editing in SpaceLand." MIT EECS Undergraduate Advanced Project. May 2007.

Van Waveren, J.M.P., "From Quaternion to Matrix and Back." 2005, Id Software, Inc. 8 May 2011. http://cache-www.intel.com/cd/00/00/29/37/293748_293748.pdf

"Verlet Integration." Wikipedia, the free encyclopedia. 8 May 2011 http://en.wikipedia.org/wiki/Verlet_integration

Wendel, Daniel. "Designing and Editing 2.5-Dimentional Terrain in StarLogo TNG." MIT Master of Engineering Thesis. August 2006.