Blurred Vision:

Autonomous Characters with Cinematic Camera Control

Ben Resner, Bill Tomlinson, Hao Yan
MIT Media Laboratory
{benres, badger, yanhao}@media.mit.edu

ABSTRACT

In this project, we describe the design and implementation of an autonomous character system with special cinematic camera control. A 3D autonomous character (a dog) is made to wander in a 3D scene hunting another 3D character (a rabbit). The dog is equipped with a virtual nose to locate the rabbit. Value-based behavior control methods are employed for modeling the autonomy of the characters. A motor system with simple collision control is implemented for moving the characters, switching animations, and avoiding colliding with scene objects and boundaries. To achieve cinematic effect in scene rendering, a synthetic camera automatically alternates between an interesting third person view of the scene and first person views from the dog's and rabbit's perspective. We also implemented depth of field effects such that only a slice of the Z-depth is in perfect focus. Other related graphics technologies we tackle are inverse kinematics and texture mapping. The system is implemented in Java with AGT animation file format and runs on Windows NT.

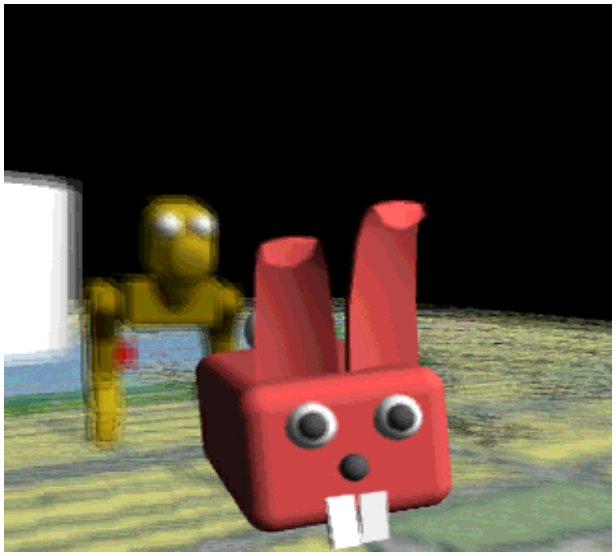

Figure 0: The rabbit being chased by the dog.

1. INTRODUCTION

In this project, we design and implement an autonomous character system with cinematic camera control.

We?ve made an autonomous dog that hunts for a rabbit. A synthetic camera chooses a variety of shots to effectively convey the events that happen during the chase. The dog and rabbit exist on a street map downloaded from Mapquest.com. The rabbit is constrained to follow the streets. It can make sharp turns and move fast. The dog is not constrained by the map, but must avoid randomly placed obstacles. Simple bounding box collision is used for obstacle avoidance. Also, the dog cannot make sharp turns.

When far away from the rabbit, the dog searches for scent trails the rabbit leaves behind. When the dog is proximate to the rabbit, it directly accesses the rabbit's location and pursues in a straight line.

In order to show off the interaction between the dog and the rabbit, we built an automatic cinematography system to decide on the position and orientation of the camera at every clock tick. Traditional film cinematography has a lot to offer in this respect. We tried to draw on the heritage of film, modeling the cinematography of the scene after the speeder bike chase scene from "The Return of the Jedi."

To focus viewers? attention of the most important element of each shot, we implemented depth of field. Depth of field is a powerful cinematic technique which adds a dimension of realism to our work. Traditional first person render viewpoints show all objects -- near and distant -- fully in focus. With depth of field, only a portion of the world in the camera?s view frustum will be sharply rendered. Changes in the shot show by the camera, coupled with depth of field changes, will produce a dramatic sense of shifting attention. This should help the user form an emotional connection to the characters.

We're using Java user interface controls to allow the user to modify scene parameters in real time. The user can control the speed of the rabbit and dog, and the dog's turning radius. This allows users to experiment with values to attain optimal dramatic timing between dog and rabbit.

The scene and characters were modeled and animated in 3D Studio Max. The animations and geometry were exported using the AGT exporter from Animetix Technologies. The dog model was animated using the "bones" system of inverse kinematics, with occasional hand touch-ups. Care had to be taken in the model to avoid unnecessary transforms that can confuse the scene-graph.

In the next section, we briefly describe the goals of this project. We then describe in detail the design and implementation of important system components. The description of our achievements and lessons learned follow. We then list the contributions to the project by each group member. We conclude this report by proposing some interesting future work.

2. GOALS

While planning this project, we had several main goals with respect to both the process and the product. We wanted to design a course of implementation that would allow each of us to learn at least one area that we were not familiar with. We also wanted to produce a product that worked well as a unified whole.

We chose the dog and rabbit project because it had room for the three main areas of work we wanted to pursue: autonomous behavior, animation and cinematography. On the behavior front, Hao needed to learn how to design an autonomous system that could respond appropriately to an ever-changing environment. With regard to animation, Ben needed to learn how to shepherd an animation all the way from 3D Studio Max until it appears on the screen in our graphics system. Bill needed to figure out how

to apply his background in cinematography to the world of computer graphics, specifically choosing the depth of field problem as one that would force him to dig all the way to the bottom of the graphics engine.

Since each of us has strengths that the others lack, we chose a project that would cause us to work together fairly closely, so that we could leverage each other?s skills if we got stuck. This worked out quite well, with each of us having something to bring to the table.

3. SYSTEM DESIGN AND IMPLEMENTATION

In this section, we describe the design of important system components, including Character Design, Environment Design, and Cinematic Camera Control.

3.1 Character Design

Having nice 3D character models and animations is not enough to make a compelling autonomous character. To achieve the autonomy of the character, we need a mechanism that decides what to do under certain conditions. Following [Blumberg & Galyean, 1995], we built a simple value-based behavior control system for our characters. A motor system, which provides the behavior system with high-level commands such as MoveForward, PlayAnimation, etc., is built separately such that it minimizes the amount of "house-keeping" required of the behavior system.

3.2 Behavior system

It is helpful if we take the "Intentional Stance" [Dennett, 1987] of the characters when design them. Thinking about a complex system by means of the intentional stance suggests that the most effective way to understand that system is by means of attributing intentions, drives, motivations and desires to it. All characters have certain desires (such as a desire to eat or drink), which arise from their internal needs such as hunger and thirst. They can take actions to fulfil their desires such as finding water when thirsty. They also have certain beliefs. For example, the beliefs could be "drinking water satisfies my thirst", "water can be found in a river", etc.

The behavior system is modeled as a finite-state machine, in which each state represents a resulting state after performing an action. Inside a state, the system constantly updates the value of those desire and state variables. When one or more values come across certain thresholds, the system performs some actions to change to another state. Figure 1 shows the simple behavior system we implemented for the dog.

In this behavior system, the dog only has one desire, which is finding the rabbit and catch it. Several variables are used to keep track of the dog?s perception about the world and its own condition, including:

- Fatigue level, updated each frame by different increasing rate in different states
- Smell value being picked up from the environment, updated each frame
- Distance from the rabbit, updated each frame

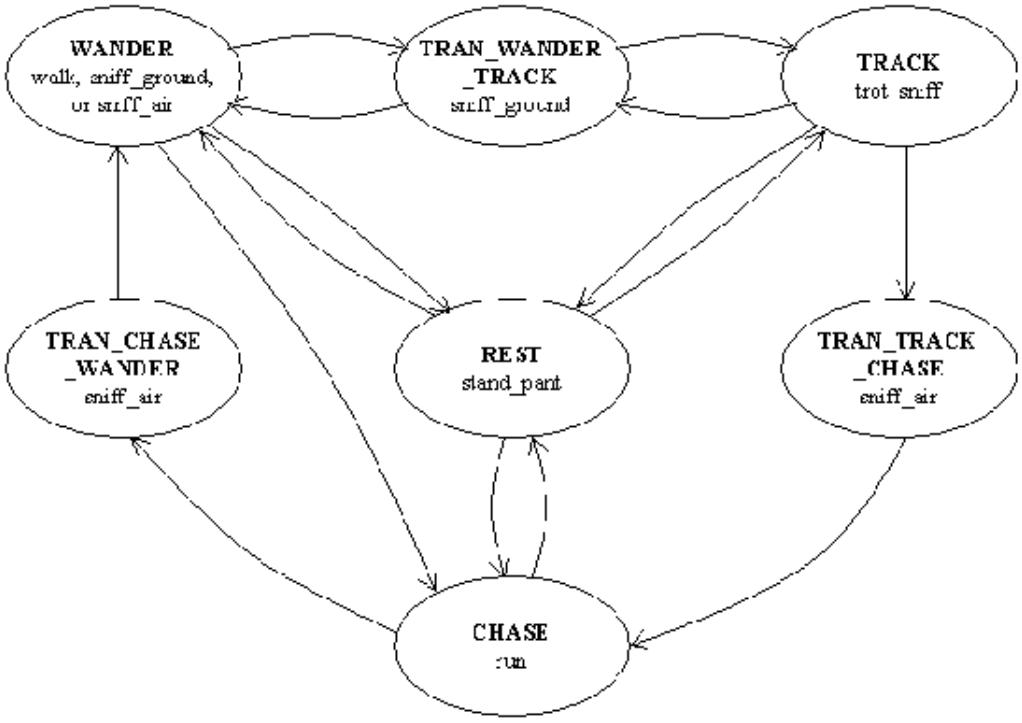

Figure 1: Behavior system for the dog

The four major states and three transitional states along with those transitional actions define all the behaviors that the dog has. For example, in the WANDER state, if the smell value picked up is greater than the sensitivity value of the dog?s nose, the dog will transit into the TRAN_WANDER_TRACK and then TRACK state. The resulting behavior will be: if the dog come across a smell path left by the rabbit, it will sniff the ground for a while and then start tracking the smell path. Appendix A listed state transitions for the dog behavior system.

The behavior system for the rabbit is much simpler. It just has two states, CHASED and NOT_CHASED. The state transition happens when the dog comes into or out of the rabbit?s visual range. Figure 2 shows the rabbit?s behavior system.

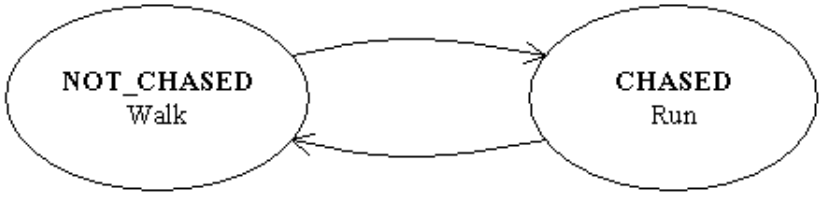

Figure 2: Behavior system for the rabbit

3.3 Motor System

The goal of the motor system is to insulate the behavior system from the graphics system. Where the graphics system requires an update each and every frame, the behavior system only needs to update at decision points such as proximity to the rabbit, or unacceptable fatigue. In essence, the motor system should take as it’s input natural language like commands such as "run away" or "pant for ten seconds".

The motor system also ensures motions are physically correct. If the behavior system commands the dog to turn around, the response should not be an immediate snap. The motor system ensures the turn is smooth.

Finally, the motor system inserts transition animations from one sequence to the next. This allows the animation to blend from run to stand by inserting a "run to stand" animation. Currently, the system must wait until the end of an animation to do this. Ideally, the animation could be interrupted and blended from one state to another. This is very important for interactive graphics where a user could provide input at any point.

3.4 Collision Detection

A simple collision control mechanism is built into the motor system to avoid the character moving across an object in the scene or other characters, and moving beyond the boundary of the environment. Basically, no matter what the shape of the character or object is, they are treated as cylinders. When the motor system detects the character is in the vicinity of an object, it starts to add a slight angle to the moving direction, based on the angle (theta) between the character?s moving direction and the direction heading the object. Currently, we are using the following updating equation:

Direction? = (1 + alpha * cos(theta))Direction.

In the above equation, Direction? and Direction are angles, and alpha is an arbitrary made weight. Figure 3 shows the object collision control mechanism.

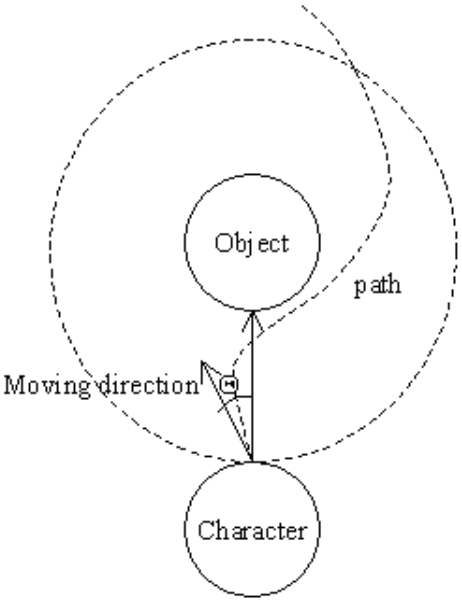

Figure 3: Object Collision control

The boundary control is based on similar mechanism.

3.5 Character and Animation Modeling

We used 3D Studio Max to model the dog. Inverse kinematics is used to simplify the process of creating the animations. While the rabbit is very simple and cartoon-like, time was spent making the dog model anatomically correct.

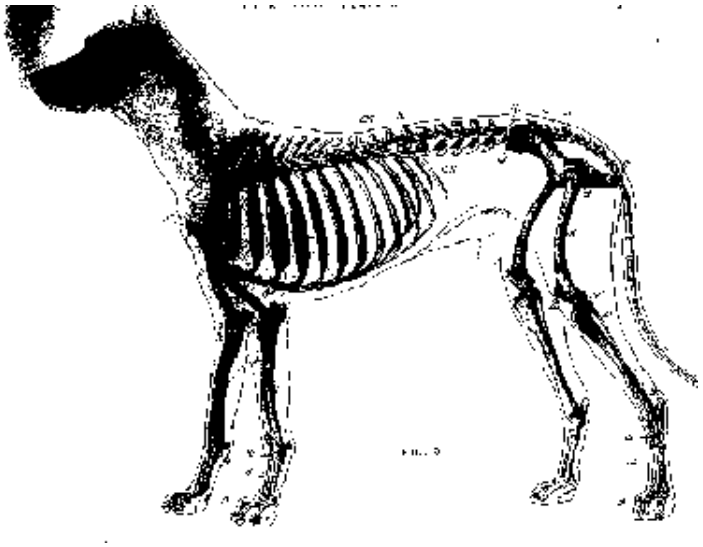

Figure 4: Anatomical picture of a dog

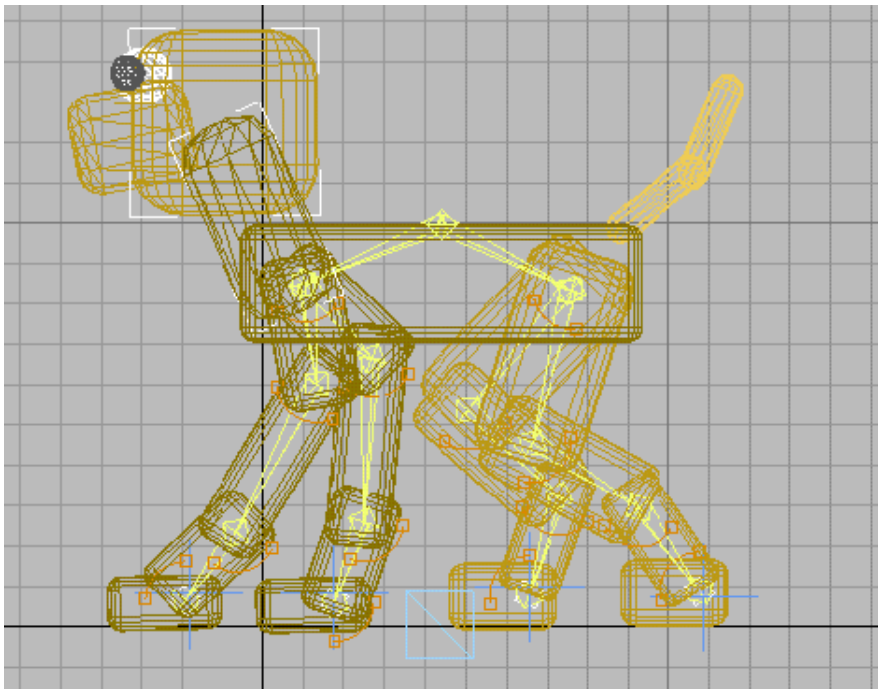

Figure 5: Our version in 3D Studio Max. The yellow represents the skeleton, and the orange arcs are the joint constrains. The blue crosses at the feet are "grab" points for positioning the feet and having the system solve a correct inverse kinematic solution. The blue box at the bottom is the top level un-animated node.

3.6 Graphics rendering system

3D data from 3DSM is exported from 3DSM using the Gamut-DXM exporter from Animetix. This creates .AGT files, which are similar to VRML and Open Inventor in that they are a text-based scene graph representation. These files are read in by the AGT reader, and rendered using the scene graph viewer written by Marc Downie and Dr. Bruce Blumberg. (see Appendix A) This system uses OpenGL as it's base rendering system.

The modeling must be completed before any animation can take place. Adding geometry to the model necessitates tediously adding that geometry to each animation and ensuring it's properly integrated. This typically leads to bugs. Great care was taken to ensure the model was acceptable before any animation could occur.

Each model contains a top-level non-animated node that represents its center. This point is used to align objects to each other, and align the creatures to the world. It's essential to make sure there are no animations on this node.

3.7 Environment Design

The environment is a flat plane with a streetmap of the MIT-Cambridge area texture mapped on its surface. Furthermore, obstacles hover over the surface which are low enough to block the dog, but high enough to let the rabbit pass. Rather than model these obstacles with the scene, they are each separate

objects. This allows us to dynamically create the scene.

3.8 Texture Mapping

Smell is represented visually by directly modifying the ground texture map via a call in OpenGL. Specifically, the rabbit leaves behind a trail that represents the residual odor of its passage. This gives the user direct visual feedback as to the dog's ability to follow the trail. A user interface button turns off texture mapping.

Texture maps are typically stored on the graphics card to increase performance. Touching the texture map each frame, however, causes the texture information to have to be pushed onto the bus each frame. This results in performance degradation. A copy of the texture map is kept in RAM such that we only need to push the texture map onto the graphics card, and don't need to ever pull it back. Turning off the smell texture mapping increases performance.

3.9 Cinematic Camera Control

In order to show off the behavior of autonomous characters, it is important to have some way of selecting the viewpoint from which the scene will be displayed. Film makers confronted this problem a hundred years ago when they started to put people in front of their cameras. Since then, a thorough grammar has been developed for conveying events, emotions and relationships. This grammar utilizes the two main tools of the cinematographer?s trade - camera and lights - to direct observers? attention and thereby control their perception of the scene.

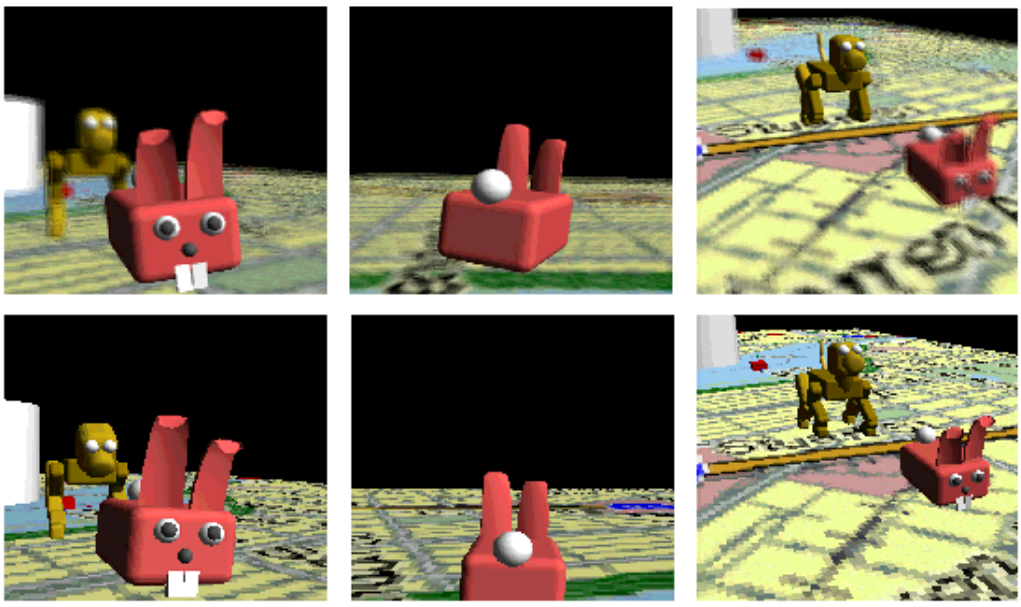

Figure 6: Three types of shot from our demonstration. The top row has depth of field turned on.

In this demonstration, we?ve tried to convey the relationship between the dog and the rabbit. There are three main shots that capture the interaction between the two creatures. Two of the shots show both characters (a two-shot), with one focusing on the dog and the other focusing on the rabbit. The third shot is from the dog?s point of view. By intercutting these three shots, it becomes possible to convey the chase more effectively than any single shot could.

In order to implement these shots, we pass the position of the two animals to a third autonomous creature, the CameraCreature, at every clock tick. This creature then decides where the camera should be placed, depending on which shot is currently active. The CameraCreature chooses its position based on a character-specific algorithm. It calculates its own position based on either the position of the dog or the rabbit, or based on the axis between the two characters. This axis has an analog in traditional cinematography, were it is referred to as an *eye-line*. It is conventional to stay on the same side of an eye line during each sequence, in order to preserve the screen direction of the main characters from shot to shot. This is called the "180 degree rule." (Malkiewicz, 1989)

Once the camera has calculated its position and orientation in world coordinates, it passes these values to the graphics system, which renders the scene from that viewpoint.

One of the main graphics problems we had to tackle in the cinematography of this interaction is depth of field. Depth of field allows a portion of the action in a scene to be in sharp focus, while the rest of the world loses focus as it gets farther from the center of attention. By causing a small area of the screen to be in focus, it is possible to draw the attention of an observer to that point.

We implemented depth of field using a multi-pass rendering technique. (SIGGRAPH 97 Course Notes) We keep the focal point fixed (the point where the camera is looking), jitter the camera?s position in a range around it?s primary position, and add the result of each jitter-step to the accumulation buffer. This necessitates a substantial slowdown in our frame rate (>5x) because the scene needs to be rendered five times per clock tick, plus a bit of additional overhead (since we touch the accumulation buffer 11 times per tick).

The addition of depth of field adds greatly to the cinematic feel of our project. Allowing the dog to be a blurred shape looming in the background really draws attention to the rabbit. Letting the dog focus on the rabbit shows that it is the object of his attention. Directing the audience?s attention is the cinematographer?s job. Control over depth of field is an excellent tool.

3.10 programming language and rendering framework

We used Java as our primary programming language for it’s ease of development, and cross-platform functionality. The AGT scene graph viewer is written in a combination of Java and C++. Some functionality, such as directly writing to the texture maps, required "drilling down" into the C++ to directly access OpenGL calls.

4. ACHIEVEMENTS

We realized our first goal of creating a realistic chase scene between a rabbit and a dog. But we also created something funny. The shots and cuts of the dog and rabbit are comic. The dog demonstrates a single-minded intensity, while the rabbit has a maniac fleeing expression. The camera angles strongly reinforce these personalities.

5. LESSONS LEARNED

Interactive systems that have cinematic camera control are much more effect than those without. The character designing method we used (desired, belief, action, and the separation of motor system from behavior system) is appropriate and effective.

The largest code overlap was the behavior system written by Hao, and the motor system written by Ben. To make this collaboration work, we needed to clearly define our APIs. Source code control (Source Safe) greatly helped this effort. The motor system exposed sufficient information that Bill was able to write the camera system without additional modifications.

In particular, we learned:

- 3D Studio Max as a modeling and animation tool
- AGT exporter
- Open GL
- More comfortable with vector math as it relates to scene graphs
- More comfortable with Java, Java Native Interface (JNI) and C++
- Utility of using 3rd party graphics packages to achieve sophisticated results.
- Utility of source code control

We also learned it's much easier to talk about something than to do it. It took ten times longer than we thought to implement everything.

6. INDIVIDUAL CONTRIBUTIONS

Ben Resner 3D models, animations and exporting. Motor control system. Realtime texturemap modifications for olfaction trails.

Bill Tomlinson: Autonomous camera control. Depth of field.

Hao Yan: Design and implementation of the behavior systems. Set up class hierarchy of the characters and the environment. Implementation of collision control in the motor system. Writing code to interface with existing AGT scene graph rendering and controlling code.

7. FUTURE WORK

- Faster graphics performance
- Actual robot vision for close range object detection (can integrate technology developed by two other 6.837 students, Scott Eaton and Delphine Nain)
- More sophisticated modeling of scent trails and scent detection.
- Feet stay planted on ground during animations
- Motor system allows animations to be interrupted and implements smooth blending
- Camera collision avoidance
- Ability to interrupt animations and smoothly transition to new animation
- Interactive lighting design that changes to better illuminate each shot

8. ACKNOWLEDGEMENTS

Thanks to Professor Bruce Blumberg and Marc Downie for valuable design advice and continuous help in implementation. Thanks to Tim Bickmore for advice on implementation of the finite-state machine. Thanks to Professor Seth Teller and other teaching staff for constant feedback.

9. REFERENCES

1. B. Blumberg and T. Galyean, "Multi-level Direction of Autonomous Creatures for Real-Time Virtual Environments". In Proceedings of SIGGRAPH ?95.
2. D. Dennett, *The Intentional Stance*, MIT Press, Cambridge, MA, 1987
3. Malkiewicz, K. *Cinematography*. Simon & Schuster, New York, 1989.
4. SIGGRAPH 97 Course Notes, "Programming with OpenGL: Advanced Techniques", 1997, p.83-85

10. APPENDICES

Appendix A: External Code Resources

We used the following pre-written code for this project. In the case of the scene graph viewer, we made some small modifications to achieve our goals. The remainder of the list we used without modification

- Scene graph viewer written by Dr. Bruce Blumberg and Marc Downie.
- Linear Algebra Java library written by Michael Patrick Johnson
- OpenGL and GLUT (GL Utilities) toolkit (http://www.opengl.net)
- Animetix Gamut-DXM exporter (http://www.animetix.com)

Appendix B: Source Code Control

We used Microsoft’s "Source Safe" as our source code control system. We found it to be a useful tool in maintaining code synchronization. Our group is accustomed to using text based CVS. This project was an opportunity to investigate a graphical system for source code management.

Appendix C: State Transitions for Dog?s Behavior System

STATE_WANDER:

State_in:

play walk animation

change fatigue increasing value

initiate timer for direction duration

State_out:

Tired -> STATE_REST

Enemy visible -> STATE_CHASE

Smelled Enemy -> STATE_TRAN_WANDER_TRACK

State_update:

Collision detection

Pick an orientation randomly every 2 seconds

STATE_CHASE:

State_in:

Play run animation

Change fatigue increasing value

State_out:

Tired -> STATE_REST

Lose visual sighting -> STATE_TRAN_WANDER

State_update:

Collision detection

Change orientation to head the enemy

STATE_REST:

State_in:

Play pant animation

Change fatigue increasing value (negative)

State_out:

Relaxed -> prev_state

STATE_TRAN_CHASE_WANDER:

State_in:

play sniff_air animation

change fatigue increasing value

initiate timer for duration of the state

State_out:

timer off

State_update:

N/A

STATE_TRACK

State_in:

Play trot_and_sniff animation

Change fatigue increasing value

State_out:

Tired -> STATE_REST

Rabbit in visual range -> TRAN_TRACK_CHASE

Lose track of smell -> TRAN_WANDER_TRACK

STATE_TRAN_TRACK_CHASE

State_in:

Play sniff_air animation

Change fatigue increasing value (negative)

Set duration timer

State_out:

Timer off -> STATE_CHASE

State_update:

N/A