6.837: Computer Graphics Fall ’99

Synthetic Vision for Virtual Character Guidance

Scott M. Eaton & Delphine Nain

Abstract

At a certain point in the development of a synthetic character, the question of virtual sensing must be addressed. The advent of lightning fast processors have made it possible to move real-time sensing from theory into design. Of the senses, the most significant first step is in the design of real-time visual sensing, which has immediate applications in the A-Life field. The vision system detailed in this paper provides first pass motion tracking in three dimensions with reliable accuracy. Built around the Gl4Java graphics library, this real-time vision system, designed for implementation into a virtual character, uses consecutive framebuffer analysis, along with a character’s implicit knowledge of his bodily motion, to extract object motion from the field of view.
Top.

Introduction

The Synthetic Characters research group at the MIT Media Lab is focused on the construction of characters that have the ability too react autonomously to changing states of their environment and user interaction. These reactions draw upon a robust underlying behavior architecture that maintains the current state of a characters goals, motivations, affect, and history. One test of a truly autonomous character is to take him and place him in a virtual world, of which he has no knowledge, and see if he reacts in a way that is consistent with what the user knows of the characters motivations, drives, and personality. To successfully adapt to this new world a character must be given "sensing" capabilities that allow him to gain knowledge of his new surroundings. This paper addresses the challenges associated with synthetic vision in virtual character.

The challenges of synthetic vision run parallel to the difficult problems of machine vision that have been addressed in the robotic vision community by Horn [1] and others. Specifically, how do you extract knowledge of the environment from a given set of camera frames. This problem quickly becomes difficult in the "real world" when scene complexity increases or when environmental noise, imperfect optics, and camera motion begin to influence the images. Even the best machine vision systems that exist today can only extract a fraction of the information that is available to biologically based creatures. Because of the difficulty, vision systems must be designed around their intended use, thereby eliminating potentially expensive extraneous calculations. Similarly, the scope of a vision system in an artificial world can quickly become computationally unmanageable, and must be tailored for it intended use. The system constructed by the authors and was designed to provide only motion information to the autonomous creature. By analyzing changes in his field of view after the relative camera motion is subtracted, the creature is able to update his position to track the moving object.

This vision system leverages many simplifications afforded by using computer graphics that are not possible in real world vision analysis. First, we assume that the autonomous creature has knowledge of the magnitude of his relative changes in position and orientation, i.e. his dx, dy, dz, and deltaRotation from one frame to the next. Second, using a properly sized orthographic camera makes subtracting out the translational motion of the camera (dx, dy) from one frame to the other nearly trivial, and depth independent. What is not zero from the difference of two consecutive frames is any pixel with a motion in the world. This relieves the system of having to do optical flow calculations on consecutive framebuffers in order to find the same information. Similarly, a perspective camera is used to subtract out the motion of pixels due to rotation of the camera, again depth independent. Coordinating these two camera transformations allows the system to predict reliably the actual motion of an object in the scene along the x,y and z axis .This information is used to update the characters position and/or orientation in order to track the object.

By taking advantage of the above simplifications, we have developed a system that gives an autonomous character the ability to track moving objects in real time. Additional computation, such as feature detection (edge detection) could be added to the system, but would slow the system to non-interactive rates. The system could also be extended to track multiple moving objects in a field. This would allow the character to selectively focus his attention on one object or track the entire motion field, depending on his current behavior preferences.

Top.

Goals

The goals set at the beginning of the project, in our team proposal , were listed as follows:

1) Implement openGL viewer using gl4java:

- write an OpenGL program that displays a scene in a window
- modify that program so that it displays the scene in one window and
another view of the scene (the camera's point of view) in a second window
- modify so that if the the camera is driving around the set, the scene
in the second window is constantly changing

2) Construction of world:

- we will research and learn how to use OpenGL.
- modeling of the world, the character, the animations: simple object
primitives, model of our character, simple animations of the camera and the spotlight.

3) User Interface:

- user can select an object by clicking on it in the Viewer and move it
in 3D space with the mouse (use of one or 2 buttons).

4) Synthetic Vision:

Basics (for testing and to get comfortable with basics in vision processing):
- Figure out how to use glReadBuffer() to read the image out of the 2nd

window into Camerabuffer.
- Figure out how to modify the image out of the 2nd window and write it back into the Camerabuffer (e.g. false color a region)
- Do vision processing on results from the Camerabuffer, and then false color the independently moving object.

5) Motion Detection:

- development of an algorithm to determine which pixels are moving from one update of the Camerabuffer to another.
-extend the algorithm to differentiate the moving pixels due to the camera's motion and the moving pixels due to the object's motion.

6) Character's Behavior

- At every update, the spotlight and the camera query the motion detection system.
- the spotlight: the behavior is to rotate the axis of the spotlight so that the light emitted follows the center of gravity of the moving pixels detected by the motion tracking algorithm.
- the camera: At first the camera will not move. As an intermediate goal, the camera will move on the x-y plane to minimize its distance from the center of gravity of the moving pixels. As an ultimate goal, the camera will move in 3D to minimize its distance from the center of gravity of the moving pixels.
- collision detection: the camera has to stop within a certain distance from the moving object.

Top.

Achievements

The problems and solutions encountered throughout the course of the project are listed below in context of the goals detailed above.

1) Implement openGL viewer using gl4java:

Our programming platform was java 1.1 and gl4java . The applet instances three classes that extend the GlAnimCanvas (the Java instantiation of the GL window) which serve as the openGL windows for displaying the world view, the orthographic camera view and the perspective camera view. Two additional GLAnimCanvases can be displayed which show the results of the buffer comparisons at every step. As independent animated canvases each instance has its own thread, yielding five threads when all windows are active. We were aware that having more than one active thread is trouble, but we didn't want to modify the base code for the gl4Java glAnimaCanvas since we were unfamiliar with it. Leaving the threads unsyncronized quickly lead to problems. When unsyncronized, each animated canvas executed its main display() method at random, causing timing problems in the buffer reads and compares that needed to be done in a cascading order. Once this was identified as the source of the error, the threads were synchronized, thereby restoring the proper data flow, i.e. a "lower" animated canvas will not be taking data from a "higher" animated, until that data has been updated to current time and is ready to give (see Figure 1).

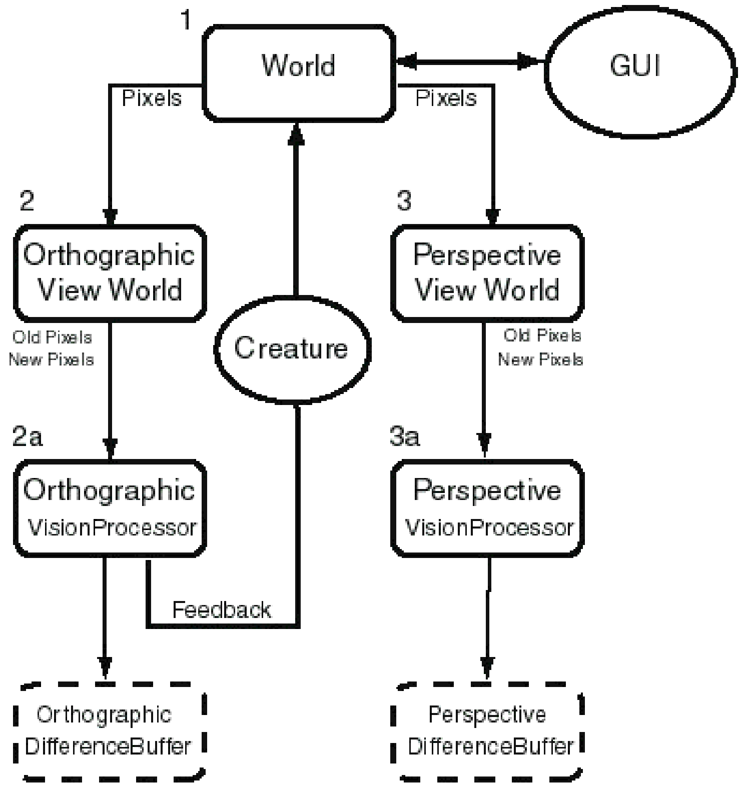
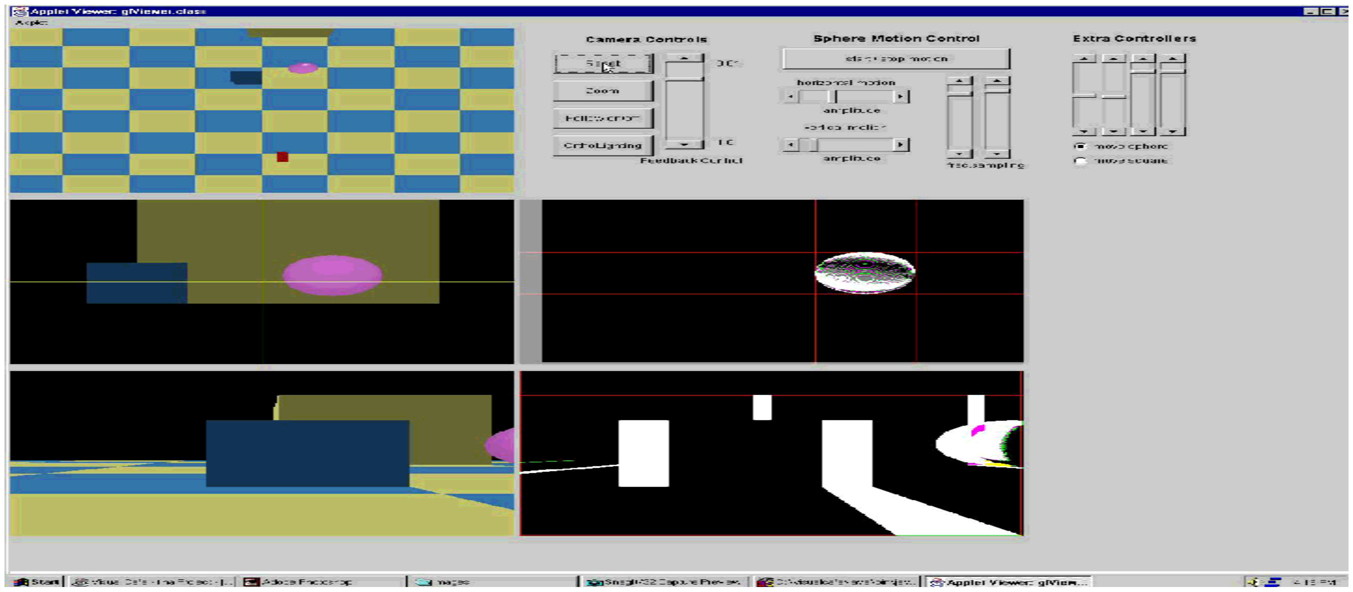


Figure 1

Figure 1. Flowchart of our vision system. Each object represents a java class.



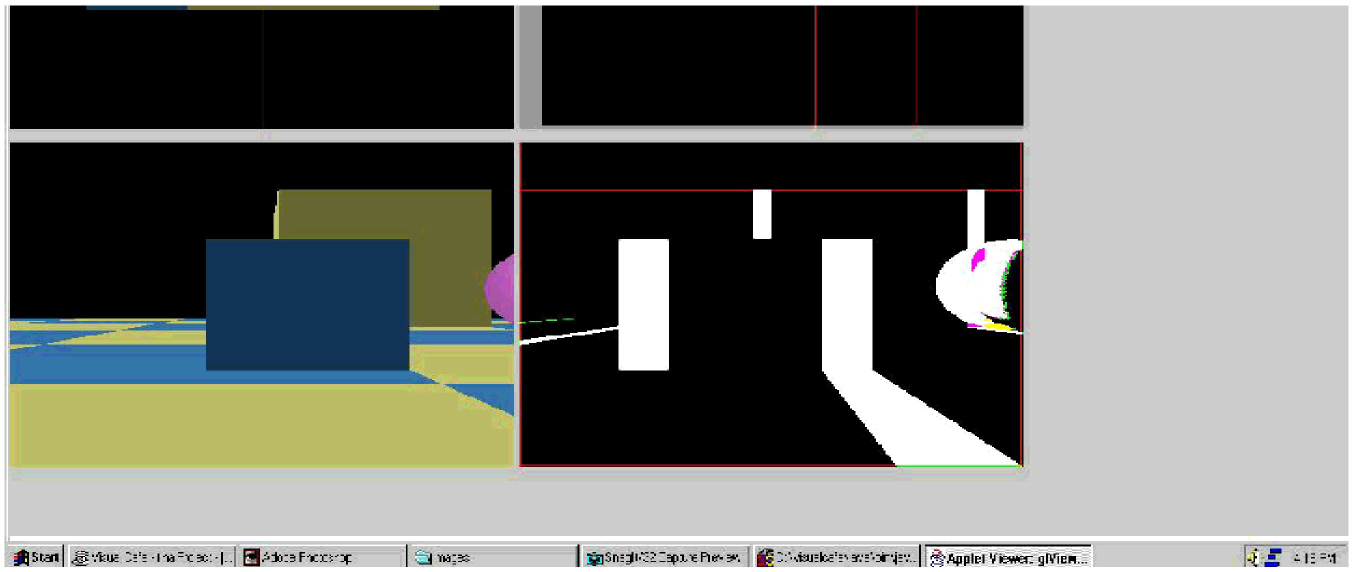


Figure 2. Screen capture of our program.

The World.class is the "highest" class, and is assigned the first position in the thread sequence. This class handles all of the gl initializations, geometry updates when objects move, and camera updates. It is required to update the position of the camera at every step. In the update() method, first the creature camera rotates and then renders from its point of view (the creature is the focal point) a perspective view of the world (see Figure 3). This perspective view is read into an array and passed to a lower animated canvas for display, the PerspectiveViewWorld.class. Then the creature camera is translated and renders from its point of view an orthographic view of the world (see Figure 4) that is likewise read into an array and passed to a lower animated canvas for display, the OrthographicViewWorld.class. Then, the method transforms the cameraMatrix to a "world" view, that is a perspective view with a focal point at a fixed location, renders the scene and swaps buffers. This world view appears as the upper left window in the display (see Figure 2). The two buffer arrays are then passed down the cascade to the next animated canvases who will 1) write them to the screen, 2) pass them on along to the VisionProcessor.class for analysis. The system makes extensive use of the read and write pixel calls, which, though slow, provide the only way to get multiple views of a common world. This gives a quick overview of the main functionality that openGL provides us.

2) Construction of world:

This aspect of the projects still needs additional attention. The world is sparsely inhabited with just enough objects to test the vision system. Without the aid of GLUT (not available for gl4java) or a VRML parser, it is difficult to get anything more than the most basic primitives into the scene. Certainly additional time could have been spent integrating a parser or modeling more complex geometry, but this would have been done at the expense of progress in the vision system. This task could have been the work of a third person in our team. If we had more time the next feature we would add to the world model would be texturing and animations. This test would certainly be significant, as the system does a pixel by pixel comparison, within certain tolerances, to see if motion has occurred. Textured surfaces would be an additional test of the systems accuracy in subtracting motion.

3) User Interface:

The user interface is another area that could be refined with additional time, though its functionality is sufficient to test the vision system. The interface is through the GUI, which has assorted buttons and sliders that control assorted properties of objects in the world. A group of radio button allow the user to select which of three objects to move in the world. These objects can then be manipulated interactively by dragging the mouse in the world viewpoint. These objects can also be move programmatically along sinusoids. Sliders control the amplitude of the motion in the x,y plane as well as the frequency of the motion. Additional sliders control the pixel sampling, which set a certain amount of tolerance in the pixel by pixel comparison.

With additional time, we would have like to implement interactive object selection and movement in the main window. This would allow the user to click and drag any number of objects in the world, not limiting the selection to the objects associated with the radio buttons.

4) Synthetic Vision:

The functionality of the synthetic vision system is provided by a combination of an orthographic view (or camera) and a perspective view (or camera).

a) Orthographic View used for detecting an object after a translation.

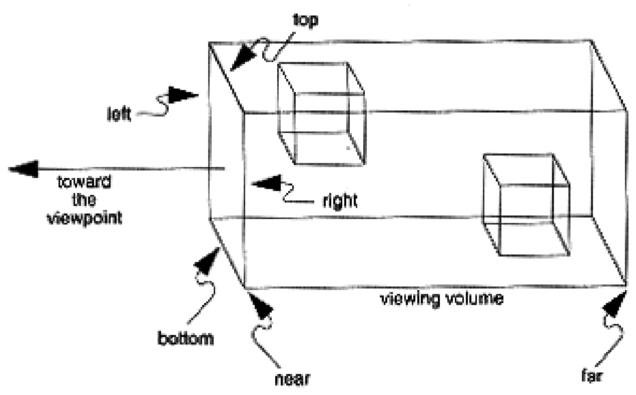


Figure 3. Orthographic view of the world. The viewpoint, or focal point is our creature camera.

As mentioned earlier, the pixels displayed on the screen by the orthographic camera are independent of depth when the camera's motion is purely translational. This means that if the translation deltas, dxws and dyws (ws - world space) of the camera are know, as it is for synthetic characters, then the apparent motion of the environment caused by camera translation can be eliminated by pixel shifting, by dxss and dyss (ss - screen space). This pixel shifting is done by the VisionProcessor.class. The instance of the class is passed two framebuffer arrays, the *old framebuffer*, oldFB (the one displayed at the last update) and the *current frame buffer*, currentFB, as well as the dxss and dyss. The visionProcessor then scans the entire array offsetting each pixel in the oldFB array dx, dy and comparing it to its corresponding pixel in the currentFB array, the results of the comparison are output into a difference buffer.

```
if  (currentFB[x-dx, y-dy ] - oldFB[x,y]) != black
then   diffFB[x,y] = white
```

b) Perspective View used for detecting an object after a rotation

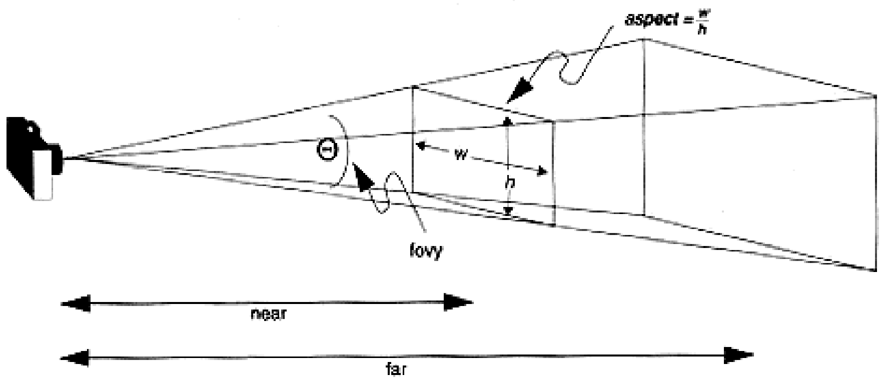


Figure 4. Perspective view of the world. The viewpoint, or focal point is our camera creature.

Similarly, the pixels displayed on the screen by the perspective camera are independent of depth when the camera's motion is purely rotational. This means that if the rotation delta is known, as it is for synthetic characters, then the apparent motion of the environment caused by camera rotation can be eliminated by the following steps. It is important to note that we chose to limit our camera's rotation to the y axis (the vertical axis in openGL), but that this algorithm can be generalized to rotation around any set of axis.

1) First we need to have the focal distance of the camera (shown as near on figure 4) in pixel coordinates.
In screen coordinates (noted with ... $_{ss}$), we know that it is the value of the near plane. When we take the camera's perspective view, we choose an angle of the field of view, Ø, the near plane distance, f, and the width and height ratio, wss/hss, that is equal to our screen ratio (480 pixels/288 pixels, for i.e.). With this information, we can calculate f $_{ss}$, the focal distance in screen coordinates:

$$f_{ss} = \text{widthss} / 2 * \tan(Ø/2);$$

2) Now that we have the focal distance f $_{ss}$, we loop through each pixel of our old framebuffer (xss,yss,f $_{ss}$) estimating where the rotation transformed it (x'ss, y'ss, z'ss) by applying the rotation matrix around the y axis.

$$\begin{aligned} x'ss &= \cos(\text{rotation}).\ (xss-(wss\ /2)) + \sin(\text{rotation}).fss \\ y'ss &= yss-(hss\ /2); \\ z'ss &= -\sin(\text{rotation}).(x1-(wss\ /2)) + \cos(\text{rotation}).f_{ss} \end{aligned}$$

note : since we start our loop at the bottom left pixel of the screen, we need to take this offset into account in our equations.
3) To make sure that our transformed pixel (x2, y2) lies on the screen, we need to scale it by (f $_{ss/z'ss}$).

$$\begin{aligned} xt &= (wss\ /2) + \text{round}((f_{ss}\ /z'ss).x'ss) \\ y_t &= (h_{ss/2)+}\ \text{round}((\ f_{ss/z'ss).y'ss}) \end{aligned}$$

4) Once the transformed pixel (x_t , y_t) is found we compare the old and current buffer as follows and the difference is outputted into a "difference buffer".

$$\text{diffFB}(x,y) = \text{currentFB}[x,y] - \text{oldFB}[xt,yt\ ]$$

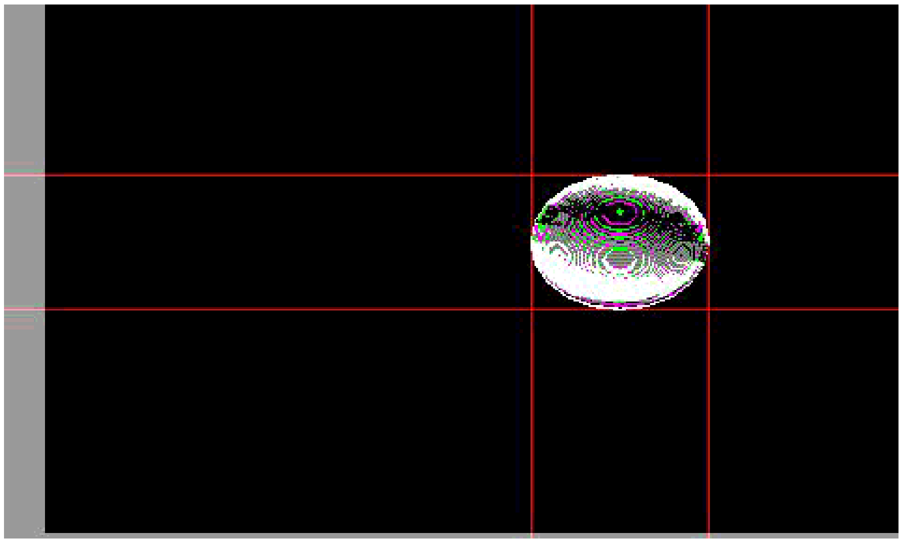
c) Difference buffer

If the pixels are the same then there is no change and they are set to black, otherwise there is a change that has resulted from absolute motion in the world and the pixel in the difference buffer is colored white. Pixels are colored gray when there is not a corresponding pixel in the other array, this is the edge effect that results from translating the entire framebuffer by dx, dy.

For various reasons, the pixel by pixel transformations may not be 100% accurate. This is caused by rounding errors when floats in world coordinates are transformed to screen coordinated and then cast to integers (i.e.pixels) and by slight variations in the shading of pixels. The system has tolerances programmed into the the visionProcessor that allows for a certain percent deviation in color value, and also a tolerance for pixel position (see Figure 8) . The later looks around at the nearest neighbors of the suspect pixel and if it is found that more than half are black pixels, then this pixel is considered noise and colored black. This is called morphology opening.

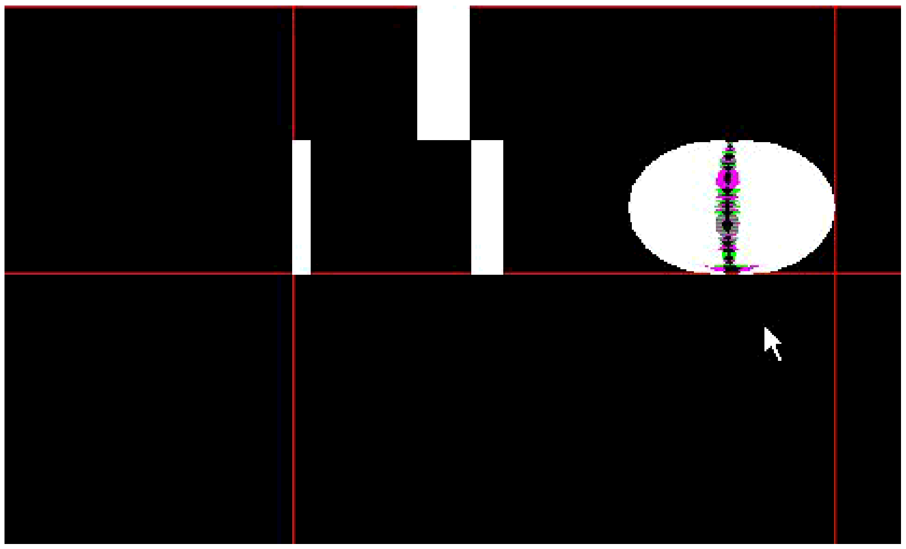
Here are the following results after the perspective view and orthographic view were visionProcessed when we rotate or translate. Clearly from these figures, we can see that orthographic view is optimal to subtract the creature's translation and the perspective view is optimal to subtract the creature's rotation.

Figure 5.



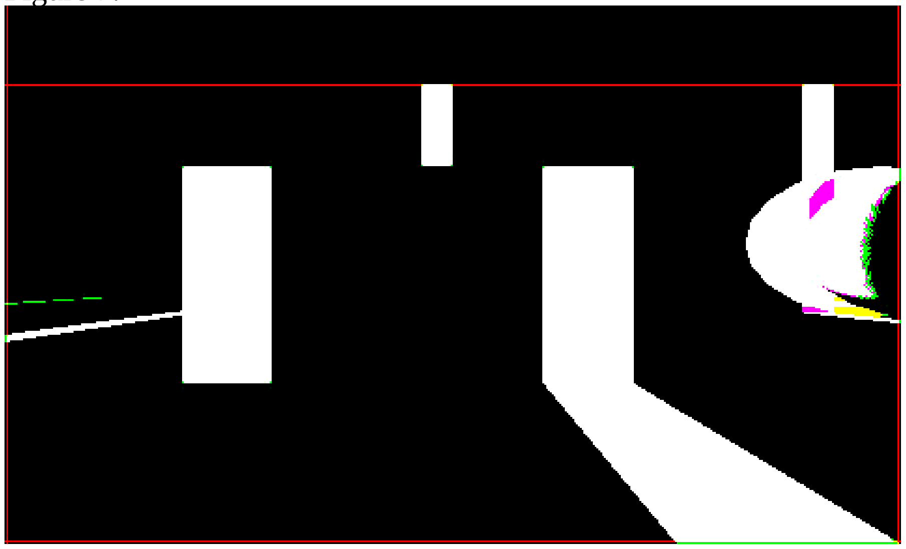
On the top left is the result of the vision processing in Orthographic mode when the creature is translating. This shows that the visionProcessor detects successfully the pixels in motion, without detecting the pixels that are fixed in the world but in motion on the screen due to the creature motion.

Figure 6.



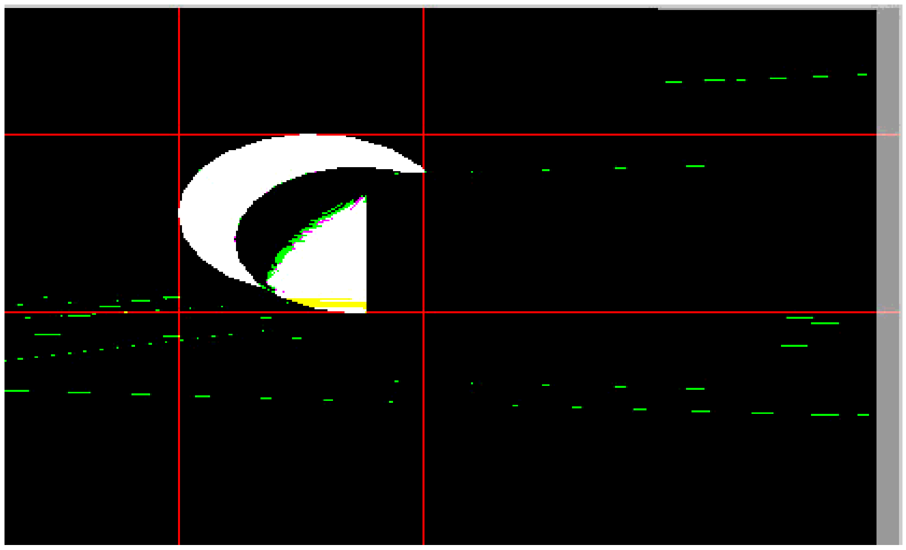
Above is the result of the vision processing in Orthographic
mode when the creature is rotating around the y axis. This shows that the
visionProcessor detects successfully the pixels in motion, but also
detects the pixels that are fixed in the world, showing apparent motion due
to the cameras rotation.

Figure 7.



Above is the result of the vision processing in Perspective
mode when the creature is translating. This shows that the visionProcessor
detects successfully the pixels in motion, but also
the pixels that are fixed in the world but in motion on the screen due
to the creature motion.

Figure 8.



Above is the result of the vision processing in Perspective
mode when the creature is rotating around the y axis. This shows that the
visionProcessor detects successfully the pixels in motion, without
detecting the pixels that are fixed in the world but in motion on the
screen due to the creature motion. The blue pixels are the pixels
removed by the color tolerance and the green ones by morphology opening

5) Motion Detection:

The motion detection at the pixel by pixel level is described above. When the difference buffer is output to the screen, it appears as one would expect, white silhouettes of the moving object's old and new positions, with any overlap in these two position colored black (see figures above). The next step in the system is to get useful information about the motion of the object out of these pixels. While the visionProcessor is looping over all x and y values in the screen doing the comparison tests, it is also keeping track of the minimum and maximum x,y values of the white pixels. The four values that result, Xmin, X max, Ymin, Ymax, are the bounding box of the pixels in motion. Taking the centroid of this box gives a target value in screen coordinates that is meaningful to the creature. This centroid value that gets updated every cycle, provides the creatures with the dx, dy values he needs to track the object in 2-D space. It is important to note here that special care needs to be to taken when converting these x,y values into world coordinates. Only for the case when the camera is pointing up/down the Z axis are these values linearly converted to world space by multiplying by the ratio of distance between clipping planes over screen dimensions.
Otherwise, if the camera has rotated, then they are some non-linear combination of the world/screen ratios, the rotation of the camera theta, and the depth of the centroid.

A more difficult problem is extracting the target depth to associate with the centroids x,y. This depth is necessary to track objects in three dimensions. In machine vision, this is generally done by comparing two images from binocular cameras and matching features between them. From there calculating the depth from the cameras is straightforward. The advantage in tracking the volume of the bounding box of the moving pixels both in orthographic (invariant volume based on depth) and perspective view (variant volume based on depth) is that the creature can compare their ratio at each update and calculate from the scaling factor whether the moving pixels are coming towards it (if scaling factor increases) or away from it (if scaling factor decreases) and move proportionally to the scale factor along its z axis. This process is not perfectly accurate but can give a good first-pass estimation of object depth.

Now, at each step, the creature has x,y,z information about where he would like to move to. This information is processed to determine what its next movement will be.

6) Character's Behavior

Given the characters current x,y,z position in the world and the target position of the moving object, our system updates the current position and rotation of the character by the difference between his current position and target position, multiplied by an error feedback parameter, which is a value between zero and one. This parameter updates the characters position to a fraction of the overall distance. This allows the character to follow quickly or to follow lazily. The best operating range was discovered to be between 0.1 and 0.2, with values up to 0.5 providing adequate results. Anything higher results in jerky follow patterns, that would be impossible for a synthetic characters motor system to handle and still look believable.

When updating with rotation as well as translation, the motion has to be subdivided into two steps, first a purely translational step, then a rotational step. If both steps were executed at the same time the orthographic and perspective cameras would be unable to give accurate information back to the character about the motion in the world.

Due to time constraints, we were not able to implement simple collision detection for the creature. Adding this functionality would help the interactivity of this applet by preventing the camera from walking inside of a primitive, thereby destroying any useful motion update data. Though, this step was not critical to the overall implementation of the vision system.

Top.

Individual Contributions

The workload was spread evenly between both members of the group. The division of labor presented in the project proposal assumed that the group would find a third member, which did not happen. This work was divided evenly between the two members. The actual break down of work is as follows:

1) Implement openGL viewer using gl4java:
Scott Eaton, Delphine Nain

2) Construction of world:
Scott Eaton

3) User Interface:
Scott Eaton

4) Synthetic Vision:
Delphine Nain

5) Motion Detection:
   Orthographic: Scott Eaton, Delphine Nain
   Perspective: Delphine Nain
   Depth: Delphine Nain

6) Character Movement:
   Scott Eaton

Top.

Lessons Learned

The first lesson that we learned was that there should be a plan in place early on for source code control. Keeping track of changes to the code made by a group member over the course of a few days was a minor problem in a group of only two people, but could be a significant problem in larger groups. Second, a solid plan for the design of the software, i.e. what classes need to be implemented, modularity of code, etc., needs to be given appropriate consideration before a significant amount of coding begins.

The learning curve for this project was very high. We learned to use OpenGL as well as gl4Java that is very similar to the java language, with OpenGL calls. The first week was spent learning about the readPixels() and DrawPixels functions, the different MATRIX modes and BUFFERS and the operations that could be performed on them. We applied what we learned in class about orthographic and perspective camera as well as transformations. Our main difficulty at the beginning was to transform our coordinates from World Space and Screen space accurately, and vice versa. One big difficulty was also to find the focal distance in screen coordinates in order to subtract the camera rotation in perspective view. At the beginning, we tried to eliminate the creature's rotation motion in orthographic view, but soon realized that it would be easier in perspective view since it is depth invariant. We were also getting the depth values of the moving pixels from the Z-buffer. Tracking the object both in perspective and orthographic mode allowed us to calculate the depth of the moving pixels directly from our vision processing data, based on volume comparisons in these two modes. Tracking the object in perspective view is also a good reality check since sometimes the orthographic view detects a moving object but the perspective view doesn't (since its field of view is much more narrow). In that case we disregard the information, since we want our creature to track objects that it can "see" in perspective mode, that is a more realistic view of the world.
One limitation of our system is that the camera moves in "steps". That is it rotates, takes a perspective and orthographic snapshot, then translates, takes a perspective and orthographic snapshot and so on... This was necessary however to separate the vision processing for rotation and translation. The use of the GUI was a great tool for debugging since we could set the motion of the objects, change the lighting, or which object to move with buttons and sliders. The display of our vision processing results (see Figure 2, middle right and bottom right panels) was also extremely useful for debugging purposes and to elaborate our vision processing algorithms. The display of the creature's point of view both in orthographic and perspective was also essential, since it is very hard to realize what happens in orthographic mode without seeing the actual buffer and being able to compare it to the perspective buffer. Another useful tool for debugging was the coloring of pixels. For example, we color the pixels that we consider "noise" due to lighting and rounding errors in green.

Top.

Acknowledgments

We would like to thank the following people for their input and guidance on the project:
Professor Bruce Blumberg, Synthetic Characters, MIT Media Lab
Yuri Ivanov, Vision&Modeling, MIT Media Lab
Damian Isla, venerable 6.837 TA, MIT LCS
Professor Seth Teller, instructor 6.837, MIT LCS. without you, we would not have done this project.

Top.

Bibliography and References

[1] B.K.P. Horn, *Robot Vision*, pp 278-293, MIT Press, Cambridge, MA 1986.
[2] D. Terzopoulos, T. Rabie, Animat Vision: Active Vision in Artificial Animals, *Journal of Computer Vision Research*, vol.1 #1, Fall 97.
[3] B. Blumberg, Old Dogs and New Tricks, Ph.D. Thesis, MIT Media Laboratory, 1996.
[4] O. Faugeras, *Three-Dimensional Computer Vision*, pp 341-393, MIT Press, Cambridge, MA 1993.
[5] M. Woo, J. Neider, T. Davis, *OpenGL Programming Guide,* Addison-Wesley Developers Press, Reading, MA, 1997.

Top.