Object Recognition In Video Games

CS674

Tony Morelli

Date Due: 12/19/2005

Date Handed In: 12/19/2005

Abstract: Recognizing objects in video games is the basis for all video game artificial intelligence. Most video game AI determines what is on the screen by the code itself. To me this presents an unfair advantage as the computer has knowledge about the states of other entities that may not be available to the average player. What is presented in this paper is an approach to object recognition in video games where the recognition device is not connected to the video game in any way. This will simulate a real human watching the television screen.

Traditional video game AI is embedded in the game itself. This allows for the AI to have access to a lot more information about the current state of the game. With all of this knowledge accessible, the AI can be tweaked to be near perfect. Although this is a clever way of doing things, it is not how a real player acts when he plays the game. A real player has no knowledge of exact positions of enemies and obstacles. A real player does not immediately know whether an object is an enemy, or an obstacle, or some kind of positive bonus. This paper will show how to determine the current state of the game without any invasion into the video game whatsoever. The game source code will not be modified in any way, and the Object Recognition System (ORS) will be responsible for watching the television screen and for controlling the game using the video game system's controller. First, the concept is presented using successive screen shots from a Nintendo Entertainment System emulator, and then the theory is applied in real time using a video camera and a XBOX video game system.

Traditional Artificial Intelligence used in video games used knowledge about the state of the game environment that was not available to the player. For example, in the Super Mario Bros game there exists a certain level where an enemy appears in the cloud. The enemy chases your player around the screen trying to throw objects at it. This enemy knows the coordinates of the controllable player because of the state of the system that exists in the memory of the running program, not because it recognized on the screen what character the user was controlling. To recognize enemies and objects using this methodology would be simple, provided the source code was

available. The situation being investigated here is a situation where the source code is not available, and the object recognition is taking place outside of the program itself. With this being the case, using traditional AI techniques will not work.

Attempting to recognize objects without having access to runtime variables seemed more like a robotics question than an AI question. Lots of research has been done in robotics vision analysis, however they all seem to be too specific. That is every object they encounter must be identified and categorized as a specific entity. They examine all entities using 3d models and transformations trying to identify what class they belong in. The examples used here could use some more details associated with them outside of their direction of travel, but the amount of details that are being considered in robotics vision research seem to be overkill. A deer in the woods is a very good example of what is trying to be accomplished here. A deer in the woods is afraid of anything that is out of the normal. Whether it be a human, a dog, or a snake, when a noise is made the deer is spooked and runs away. The deer should not be scared of most humans, or dogs, however its survival instinct allows it to escape any potentially dangerous situation. The ORS takes those natural behaviors into account while recognizing objects. Anything out of the ordinary should be considered an obstacle.

The approach used in the ORS is as follows. In the proof of concept, images were taken as consecutive screenshots from an emulation of the game Super Mario Bros running on a Linux based PC. The screen shots were saved off as a P6 pgm file. A P6 pgm file is a simple

graphic format where each pixel is represented as 3 bytes (1 byte for each of the Red, Green, and Blue color values for that particular pixel). Two consecutive images were subtracted from each other. That is, each corresponding RGB value in the second image was subtracted from the first image. Once this subtraction is complete, you are left with an array of pixels the same size as the original images that highlights the changing pixels. Any pixel in the subtracted image that is a 0 represents pixels that were identical between the first picture and the second, and anything other than a 0 represents a change in pixels, or motion. These changing pixels were grouped together into regions and rectangles were placed around them. This keeps individual objects and their locations easy to maintain.

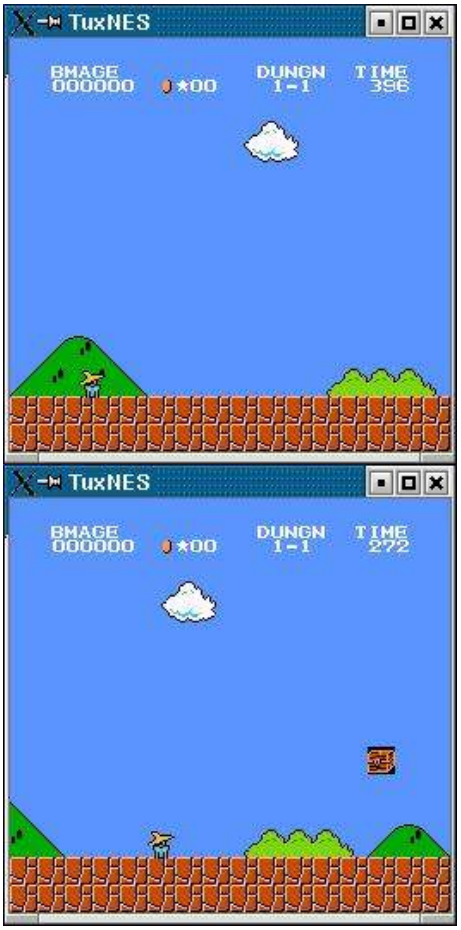

Figure 1 -- Frame 01

Figure 2 -- Frame 02

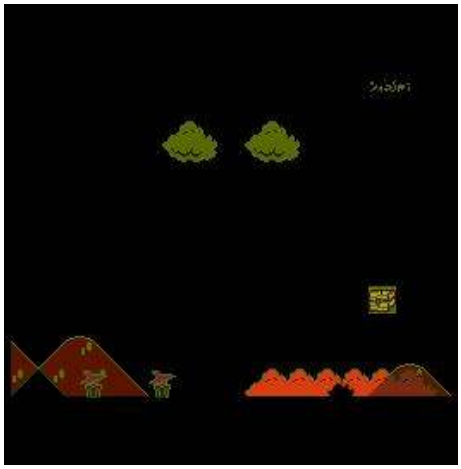

Figure 3 -- Frame 02 – Frame 01

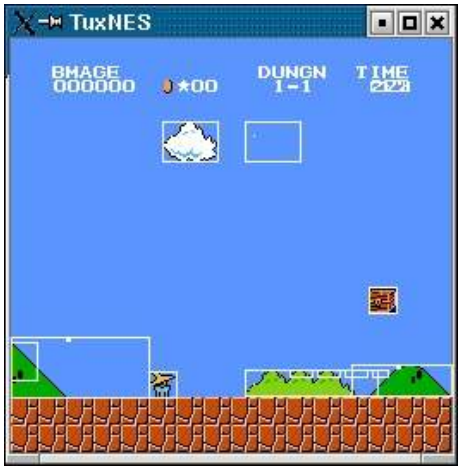

Figure 4 -- Identified regions overlaid on current frame

The resulting subtracted image easily shows the regions that have changed between the two grabbed frames. What it does not tell you is what direction the region is moving. For example, when the images are subtracted, you are left with a region representing where the object was in frame one, and a region where the object is in frame two, but it is very difficult to know which region represents frame one and which represents frame two. In order to over come this problem, a third frame was taken into account. This third frame was subtracted from the second frame leaving another images with regions identifying motion between the two frames. At this point we have 2 images that represent motion, one between the first and the second frame, and one between the second and the third frame. The regions are differentiated by their approximate location and size. A region that is the same between the 2 subtracted images identifies where the object is located when the second frame was taken. If the region is moving from right

to left in the subtracted images, the object is moving from right to left. If the region is moving from left to right, the object is moving from left to right. This is a very simple way of looking at things, but it does its job of identifying the direction of objects over a series of frames.

Now that the direction of objects is known, the identification of what object represents an enemy and what object represents the character being controlled begins. To make this determination, how a human plays video games was analyzed. When a human is playing a video game, he knows that when he pushes the directional pad to the right he expects his character to move to the right and all enemies and obstacles to come towards him. This is a strategy used in 2D side scrollers such as the Super Mario Bros game used in this experiment. Using that knowledge, it was obvious that the ORS will need that information to correctly identify enemies and obstacles. This is how humans play the game, so giving this information to the system seemed necessary. The direction parameter was added, and using that information the ORS identified the controllable character as well as the obstacles and enemies. One issue to keep in mind is that the ORS is always one frame behind the current frame. If the system is able to process the images real time this would mean the processing is taking place $1/24^{th}$ of a second after it needs to be. This should not be an issue.

The ORS was able to identify the controllable character from the obstacles and enemies. It also misidentified some items. For example, a block with a question mark is actually a good thing, but since it is moving the opposite direction as the controllable character it

is marked as an object to avoid. Also clouds and trees, which in this game are just a part of the background are marked as objects to avoid. This is not considered a failure as the goal of this project was to identify objects in the video game. Some simple AI can be added to identify what is in those objects, but for this example all objects out side of the controllable character are considered objects that need to be avoided.

One issue that could be considered a failure is the time required to do the analysis of three consecutive images. This averaged 1.2 seconds on the test computer. 1.2 seconds is a lot of time, and is too much time to ever have this accomplished in real time. Two areas of time consumption were identified: The reading/writing of the graphic files from/to the disk, and the algorithm of subtracting the pixels. The subtracting of the pixels has to be done. Each pixel must be looked at to identify motion. To make it faster, it was moved to a faster computer. As far as speeding up the disk access the determination was made that pulling frames out of memory would be faster than getting them off the disk. To achieve this, the ORS was modified to have direct access to a live camera feed pointed at a television. Outside of making it faster to read images, it also achieved the goal of having a completely independent system to recognize objects. The user would not have to manually save images, the ORS would read them directly from the stream. At this point the software (which was previously written in C++ and run on Linux) was ported to VB.NET and run on a faster computer system running Windows XP.

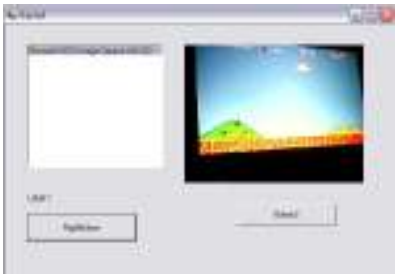

Figure 5 -- Screenshot of VB.NET App

Now having the application analyzing images directly from the live video stream increased the speed, however another aspect was lost in the conversion. Now the system had no knowledge of which direction the character was going. To overcome this, another component was added to ORS. The ORS was given the ability to control the video game itself. The Super Mario Bros game emulation was moved to run on an emulator running on an XBOX video game console. The XBOX controller was wired directly into the computer's parallel port, and the ORS would control button presses on the controller. That way when the system moved right, it would know it was moving right while processing the images.

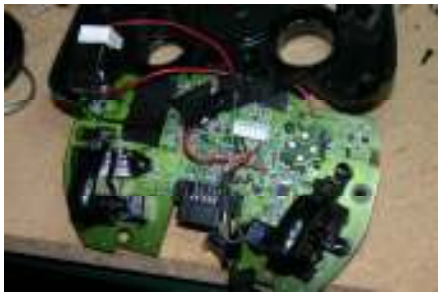

Figure 6 – Modified XBOX Controller

Figure 7 – Parallel Port/XBOX Controller Interface

The image processing algorithm needed to be modified. Doing a simple subtraction would not work any more. This is because the pixels in the lave

feed were not perfect. That is a green pixel would not show up as 100% green anymore due to reflections on the television screen, and fluctuating output from the television itself. Simply subtracting consecutive images resulted in just about every pixel having some degree of difference. A 'fudge factor' was added to determine when a pixel had changed and when it had simply looked different through the camera. All pixels that had an absolute change of more than 100 were determined to be a pixel that actually changed. The absolute change was calculated as follows. The absolute value of the difference of each of the Red, Green, and Blue components was added together to give the absolute change.

The speed of the processing went from 1.2 seconds to .75 seconds. However, this was not enough of an improvement to provide real time analysis of the images. Real time analysis was attempted, but the processing was too lagged, and the obstacle avoidance would kick in too late causing the character to be trapped by an enemy. The algorithm was working, however, and increasing the processing power of the computer running the ORS will help this substantially. The experiments were run on a Pentium M1.5 GHZ computer. Running this on a space age top of the line state of the art computer could possible bring down the computing times to allow this to happen real time. The other modification that might need to be made is instead of looking at every pixel, maybe try looking at every other pixel. Every other pixel should give enough resolution to identify regions of motion.

Overall, the algorithm discussed here for an independent object recognition system should be considered a success.

The algorithm correctly identified objects in the path of the character being controlled. The regions are identified by their coordinates, and could easily have more attributes added such as colors. With this information, the scenario presents itself nicely for an AI problem to determine which objects are enemies, which are good for the character, and which can be ignored as they are part of the background. Having the system independently control the video game also adds to the main feature of this approach, which is being completely independent of the game being played.

Sources

H. Kwon, Y. Yoon, J. B. Park and A. C. Kak, "**Person Tracking with a Mobile Robot using Two Uncalibrated Independently Moving Cameras**," to appear in Proceedings of the 2005 IEEE International Conference on Robotics and Automation.

Guilherme DeSouza and A. C. Kak, "**Vision for Mobile Robot Navigation: A Survey**," IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 237-267, February 2002.

A. C. Kak and A. Kosaka, "**Multisensor Fusion for Sensory Intelligence in Robotics**," Proceedings of the Workshop on the Foundations of Information/Decision Fusion and Applications to Engineering Problems," (INVITED TALK) Sponsored jointly by Department of Energy, Office of Naval Research, and National Science Foundation, August 7-9, 1996, Washington DC.

A description of some experiments we have done with the Quakebot to test how human it behaves and whether we can modify some simple parameters to change skill levels: Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot. This appeared in the AAAI 2000 Fall Symposium Series: Simulating Human Agents, November 2000.

Magerko, B. "A Proposal for an Interactive Drama Architecture", AAAI 2002 Spring
Symposium Series: Artificial Intelligence and Interactive Entertainment, March 2002.

Comment [1]: <!--[endif]-->