Optimized Spatial Point Detection for Computer Vision in Gaming

Chance Lyon, Noah Hopson –Walker, Adam Demers

DigiPen Institute of Technology
Redmond, Washington, USA

E-mail: {clyon|ademers|nhopsonw}@digipen.edu

Abstract
In this paper, we present a novel methodology for controlling games without the need of expensive and dedicated image processing hardware. We present an alternative to the current spatial control schemes by using point-light detection with a webcam. Our framework combines time-based image differencing, window-sum searching, and history-based point-culling to identify the correct object in a scene. These techniques allow detection of points in a scene, spatial positioning in three dimensions, and easily extrapolated acceleration and velocity data. We created a controller that proved efficient functionality similar in ways to the Sony EyeToy [7] or the Nintendo Wiimote [5]. Unlike the EyeToy, our application is independent of background noise, as it does not depend exclusively on optical flow [3] or other time-based difference algorithms. Additionally, unlike the Wiimote, our controller is inexpensive and replaceable, while providing a good deal of functionality.

Introduction
Computer Vision has been evolving recently in the computer gaming sector. Cameras have been added to many devices as a peripheral controller of some sort, going back to the GameBoy Camera in 1998. With the recent developments of the Sony EyeToy [7] (Figure 1) games, the Nintendo DS with a touch screen, and the Nintendo Wii with an accelerometer-based controller, it is clear that gamers want different kinds of ways to play games. Intuitively, the natural tendency of a game player is to physically move the body attempting to control the game.

In this paper, we present an alternative to the current spatial control schemes by using point-light detection with a common webcam. The necessary hardware consists of a common webcam and an object to support the lights that we want to detect in the application. Our framework combines a number of techniques such as time-based image differencing, window-sum searching, and history-based point-culling to identify the correct object in the scene each frame. These techniques allow for not only detection of the points in the scene, but also allows for spatial positioning in three dimensions, and easily extrapolated acceleration and velocity data.

With all this information coming from a very simple object, it could prove an inexpensive and fun way to make computer-vision based games that people can enjoy on the PC or console. One of the large limitations of these kinds of games is that they require expensive and specialized hardware [4].

Figure 1 The EyeToy from Sony

Hardware
There were some careful considerations when choosing the hardware we required. We needed a camera to supply us with a video feed. The camera needed to be able to sustain a playable frame rate at a reasonable resolution for image processing, with as few artifacts and degradations as possible. For this we chose the Logitech Orbit MP. While the picture quality is exceptional, the camera is only able to output 15 frames per second as it was designed primarily for low frame rate applications (such as an internet webcam).

Second was the decision on what physical device to use to control the game. By comparing different projects and applications [5, 7, 3], we found that most of them used simple methods for image

detection, relying solely on movement detection other than object recognition. The EyeToy (Figure 1) [7], for example, uses simple time-difference motion detection, accomplished by bit-masking the image each frame to detect movement. Similarly, another project attempted to control a labyrinth game used a webcam to sense tilt (Figure 2) by examining the environment [3]. Instead of a simple image-difference algorithm like the EyeToy, it used image flow to detect movement. Even though it is a much more robust approach, we wanted a more useful tool that has the ability to correctly detect the movement of a specified object.

Figure 2 Tilt Sensor on a Tablet PC

The Nintendo Wiimote (Figure 3) is close to what we wanted, and is capable of spatial detection as well as acceleration readings. Unfortunately, it is a very specialized and expensive piece of hardware, designed exclusively for the Nintendo Wii.

Figure 3 The Nintendo Wii Remote

We chose the paddle shape (Figure 4), a staple of classic games. To make our job easier, we decided to circumvent any complex shape-detection algorithms. Our assumptions were that if we can detect the endpoints of the paddle, we can recreate it in software. So we added a bright red LED on each end of the paddle, and chose to develop algorithms to detect the endpoints' positions, and pass them off to the game.

Figure 4 Our Device

There are a few key advantages to this approach. First, using a bright light makes the detection largely lighting and noise independent, which is a decided advantage over existing computer vision implementations [4, 6, 7]. Second, the brightness of each light and distance between the lights can be used for detection not only in 2D but 3D should the need arise.

One of the interesting issues we encountered was how the camera perceives color. While the LED's may look like a very solid red to the human eye, it can clearly be seen in this image that to the camera, they are very white with a red outline. This effect was unexpected and initially caused problems in the detection algorithm. The camera's exposure time caused bleaching in the color buffer to occur due to the intensity of the lights.

Framework Architecture

We needed a framework that allows us to rapidly prototype our design, and implement some test cases to see how it performed. C# was an easy choice, given its ease of UI design, integrated garbage collector, and built-in rendering controls. Having GDI+ immediately available was very handy to visualize our algorithms performance.

The next challenge in developing something that we could immediately use was to figure out a way to interface with the hardware. This took quite a bit of research and trial and error testing to figure out a convenient and fast way to capture raw images from the camera. We eventually settled on a wrapper around the avicap32.dll library [2]. It allowed us to do streaming video and provided a callback to get the data each frame (compared to some libraries that only displayed the video without giving you access to it). C# also allowed us to do some easy Bitmap operations intrinsically. Such as stretching the image, and saving and loading from files. This enabled us to add a simple but useful playback functionality to demo the project.

Point Detection Algorithm

Once an image has been captured by the camera, it is sent to the image processor. The entire detection process is done in a single pass over the image for

speed purposes (a 400x300 image is 120000 pixels per frame, thus the need for optimization). A full-scene pass is necessary as we cannot be guaranteed the wand will not move quickly across the scene. A single full pass also maintains a steady process time.

The algorithm is split up into three main steps: First, use temporal differences to eliminate known background pixels [1]. Second, identify points of interest that might be our LED's. Last, decide which two points are most likely to be the LED's by culling any points that seem unlikely.

We knew that to make this work in real time using software on the CPU we had to put all our processing of the raw data into one pass (or less), because having to search through 120000 pixels multiple times will begin to encroach upon the time necessary for the rest of the application (game logic, physics, etc). Our initial goal given the lack of dedicated hardware or threading was ten frames per second. So to reduce searching time we started with the idea of using a simple window search to categorize regions at a time.

The first step was to extract moving objects from the background. Thus, a calibration image is shot right when the program begins, to rule out anything that will likely remain still in the image. This stored image is compared within a threshold (to account for subtle variations and noise in the image) to new images taken during the course of the application. Any pixels too closely related to the calibration image are thrown out. This greatly reduces the search space of the algorithm in the next two steps.

While this technique is excellent in controlled conditions, it does have drawbacks. Dramatically changing lighting, background noise (like people walking by), or having the wand in the frame to begin with nullify any benefits gained.

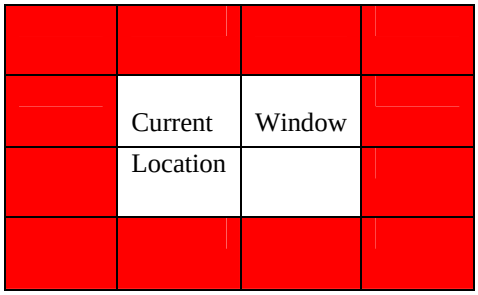

Figure 5 Color Window Representation

In the second step, we detect possible points in the image that might be our LED's. Since we know we are working with a very bright light we know we can make some assumptions about what we are going to be getting as input. The first is that we know that the red LED light will have a gradient falloff as we move away from the center of the LED. Thus we made the search window (Figure 5) look for the color with some offset to increase out chances of finding them. We also added test conditions to throw away any data outside the specific color range we are looking for. That way we are eliminating the white light and predominantly green and blue colors. Given the window, we take a sum of all pixels that pass the tests, and if they are above a certain threshold (50-70%) of the maximum possible, we accept the region as a point detected.

Our original settings produced unreliable results, with color values for the background changing more than expected. We noticed that the camera itself did its own image processing in the background. The gain, exposure, and brightness settings on the camera greatly affected the results of the algorithm. Adjusting these values to the proper settings solved this problem. Notice the change in coloration in the next two images (Figure 6 and Figure 7).

Figure 6 Tinted Exposure Settings

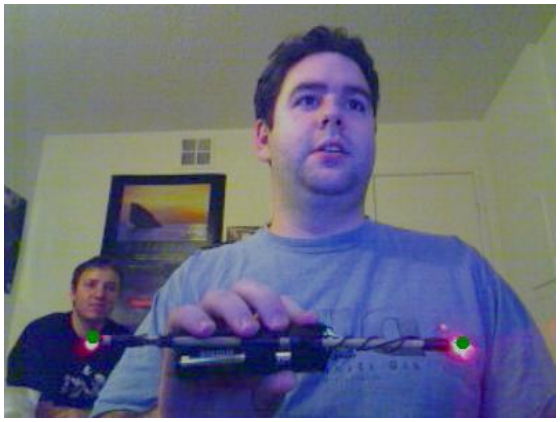

Figure 7 Even Exposure Settings

The final part of the process decides which two points created by the first step are most relevant. No one metric sufficed, so we combined several simple algorithms to cull the points. The next

image (Figure 8) shows what a raw image might look like:

Figure 8 Unfiltered Image

Each green circle on the image is a potential point that satisfies the color requirements by the first step. The ends of the wand are detected. Unfortunately, so are many other data points on the hand and face. We needed an algorithm to differentiate between relevant and irrelevant data.

The first step of the culling process eliminates groups of points too close to each other (Figure 9). We compare the distance of each point to. Pseudocode follows:

```
For Each Point in PointList
  For Each OtherPoint in PointList
    If Difference < Threshold
      Remove OtherPoint
    EndIf
  EndFor
EndFor
```

Figure 9 Image with Clusters Removed

Now we have evenly spaced regions of interest identified. We still have the LED's captured, but there is still some extra data in the list. While the history is only one deep (only the last frame is tracked), it is sufficient for removing this extra data. Given the history of the last successful points found, we can now rule out any points that are too far away from the last found points. The assumption in this step is that the player will not be moving the wand very far per frame, thus any points too far removed can be eliminated. Some Pseudo code for this process is as follows:

```
For Each Point in History
  For Each Point
    If Diff < Threshold
      If Color is closer to Ideal
        Choose Point
      EndIf
    EndIf
  EndFor
EndFor
```

Figure 10 History Filtering Applied

Finally we see something we can work with (Figure 10). We pass this data to the game logic, and record the new history points.

Now, the images shown have been ideal, in that there actually were two points on the screen to detect. One of the major issues we came across is that we aren't guaranteed this condition. The wand can be half or fully off of the screen as the user plays. Despite this error, we need to maintain our point history so that future detection continues to work. To combat this, we have several cases for each error condition to keep the points on the screen. We handle these cases by guaranteeing two evenly spaced points remain on the screen at all times for detection to resume when both points are detected again.

Lastly, we need initial points, a set that can be used at startup before we have identified the wand on the screen as the history to track the wand in the future. Thus when the application starts a dummy set of points are placed on the screen in a predefined location. When the wand is moved in proximity with them, it latches on and the true detection can begin. After we find the points, we pass it to the Game Logic and physics systems.

Physics Integration

After the image is processed the data is sent to a fairly standard physics system to form some kind

of object. For game we produced, we created a line object stretched between the two points that would collide with other game objects. This could easily be extended by using the two points given to be control points for any number of objects (a wand, a sword, a handle). Note that even in two dimensions we already have some sense of depth from the size of the line produced. While the example game operates in two dimensions, the transition to 3D would be relatively simple.

One optimization that needs to be considered for collision is to make the object they are controlling smoothly animate (rather than snapping when new data is acquired). Using a history of point locations, velocity and acceleration could be derived, allowing for smooth animation in-game to compensate for slow data from the webcam. This prevents clipping of the objects it will collide with.

Test Case: Bricks

The prototype project we produced is a recreation of the classic arcade game Bricks. The wand acts as the paddle (in blue), reflecting the ball (in white) in the direction that the user wants. Figure 11 shows an image of the game in action.

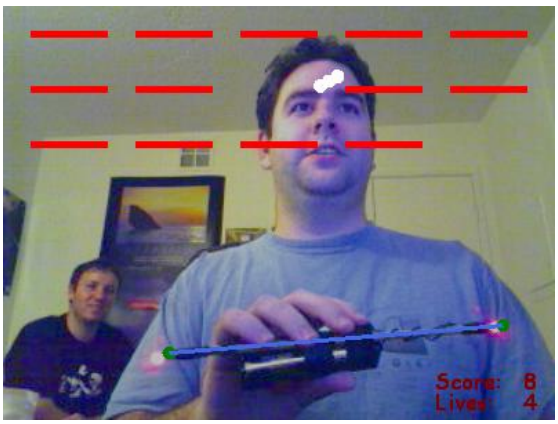

Figure 11 Game Overlay Displayed

Conclusions

Many of today's homes have webcams, and for the price of a pair of LED lights and a way to power them, a simple, fun, and highly interactive game can be produced without the need for specialized hardware like the Wiimote or dependant circuitry. It is a flexible interface providing a sufficient amount of information in a very small overhead.

Using a simple but known object for detection purposes allows for very fast and optimized algorithms to be run, and the possibilities for the number and types of game play are large. Uses from controlling our simple line, to complex movements in three dimensions and gesture recognition are easily seen. Playing musical instruments, sword fighting, aiming, steering, and fighting are all things that could easily be done with this process.

Further refinements to the algorithm can be made, depending on the type of accuracy needed in the application. Velocity and acceleration tracking to interpolate movement between frames would be useful. Keeping a deeper history of movement would aid in this. Tracking the relative intensities of each light in addition to the length between them would be useful in determining distance and tilt away from the camera if that information was needed.

Acknowledgements

Dr. Rania Hussein, for her excellent tutelage of image processing techniques, and for helping us through the publishing process.

Our families, for supporting us through the tough years at DigiPen.

References

[1] Umbaugh, Scott E. *Computer Imaging: Digital Image Analysis and Processing*. CRC Press, 2000

[2] Low, Brian "DirectX.Capture Class Library" http://www.codeproject.com/cs/media/directxcapture.asp 20, Mar 2003

[3] "Web Cam Optical Flow on a Tablet" http://www.brains-n-brawn.com/default.aspx?vDir=cameraflow 15, August 2005.

[4] Freeman, W.T., Anderson, D.B., Beardsley, P.A., Dodge, C.N., Roth, M., Weissman, C.D., Yerazunis, W.S., Kage, H., Kyuma, K., Miyake, Y.; Tanaka, K., "Computer Vision for Interactive Computer Graphics", *IEEE Computer Graphics and Applications*, Vol. 18, Issue 3, pp. 42-53, May-June 1998 (IEEE Computer Graphics and Applications)

[5] Marriott, Michael, "At the Heart of the Wii, Micron-Size Machines" http://www.nytimes.com/2006/12/21/technology/21howw.html?ex=1167714000&en=e451f00cd7b6140d&ei=5070 The New York Times, December 21, 2006.

[6] Zivkovic, Zoran. "Optical-flow-driven Gadgets for Gaming User Interface." http://staff.science.uva.nl/~zivkovic/Publications/zivkovic2004ICEC.pdf

[7] Sony Computer Entertainment Inc.: Sony Eye Toy, www.eyetoy.com, (2003)

Bibliography

Chance Lyon is a senior at DigiPen Institute of Technology graduating with a Bachelors of Computer Science in in Real-Time Interactive Simulation in April 2007. He is also working as a Junior Programmer for Zombie Studios, Inc.

Noah Hopson-Walker is a senior at DigiPen Institute of Technology graduating with a Bachelors of Computer Science in Real-Time Interactive Simulation in April 2007.

Adam Demers is a senior at DigiPen Institute of Technology graduating with a Bachelors of Computer Science in Real-Time Interactive Simulation in April 2007. He is currently working as a software engineer at Wizards of the Coast.