NanoSiege

Jeff Ohlstein
Rich Lane

December 16, 2008

1 Introduction

Our goal in creating a custom game for this project is to allow us take advantage of the potential offered by the embedded platform. We really wanted something that could be accelerated by customized hardware and couldn't run on the embedded PowerPC alone. The "Gameplay" section below describes in detail our original goal for our finished project, the "Design" section describes how our design progressed and ended up, and the "Team" section describes how our group managed itself.

2 Gameplay

NanoSiege is a simple game for two players, each controlling multiple groups of "emitters." Emitters generate a force field that both damages enemy emitters and shields them from enemy force fields. Force fields can be focused by the player, increasing damage over a small area but leaving the rest vulnerable.

Players use a mouse for control. The left mouse button changes the move target of the current emitter and the right mouse button changes the aim target. The scroll wheel changes the selected emitter group. Emitters take a weighted random walk that generally moves them to their move target. The aim target is the location towards which this emitter group will send force. The distance at the time of aiming determines the interior angle of the cone of force - clicking farther away concentrates it more but keeps the average angle the same. Force is not conserved, so if a cell is already saturated (on a per-team basis) then additional force added to that area will be lost.

Force fields are visible on the screen as their team's color with brightness dependent on the strength of the field in that location. We do not have the computational power to use a formula at each cell to calculate the amount of force present, but we can model it by simulating a large number of force particles and interpolating their effect in the space between them. This interpolation is done using a simple blending algorithm, which is physically incorrect but sufficient for gameplay and visually attractive.

Terrain adds to gameplay by providing asymmetry that skilled players can exploit. A fullscreen image and a mask are kept in memory. The mask controls whether emitters and forces can move into the cell. The final game will include several maps that can be chosen between when starting a new game. This feature did not make it into the finished project.

Background music will be played during the game. We haven't determined what music to play yet. There will be a simple menu system to allow players to select a map and start a new game. This feature did not make it in to the finished project.

3 Design

The finished game to the spec above is our hard deliverable. We have identified areas that may need acceleration and we'll spend the rest of the semester implementing the necessary hardware units. Our current partitioning handles emitters and force particles entirely in software, and force interpolation and rendering in hardware. This may change with the addition of some optional hardware outlined below.

3.1 Software

High-level gameplay is done in software. The main software components are initialization, input, emitters, forces, and the game loop. We additionally have PC-only code for graphics and input that uses SDL and fixed point trig functions. We use the preprocessor to allow us to compile the same codebase for both platforms, so gameplay changes can be quickly tested on a workstation.

The game loop executes every frame. Its primary function is to call the other components and to synchronize with hardware. Our design is currently sequential in that the hardware and software wait for each other to finish before proceeding. This is a difficult problem to parallelize because the hardware needs updated cells from the software for interpolation and rendering, and the software needs updated cells from the hardware for damage calculation. Using the second CPU and moving force particle updates into hardware are two possible optimizations, but we will wait and see how the initial implementation performs before doing this.

Emitters move in a weighted random walk towards their move target. They move multiple cells each step, which, given that they start in the same cell, leads to an attractive "digital" effect. The emitter is then damaged based on the amount of enemy force in the new cell. If the emitter's health reaches zero it dies. If the emitter is still alive it may create a new force particle. The rate at which emitters create force particles can be tweaked for appearance and performance. Force particles are given a velocity vector when created, which is randomly distributed in a sector pointed toward the aim target.

Force particles move in a straight line along their initial velocity vector. Collisions with terrain are handled by setting the particle's strength to 0, effectively destroying it. The strength of the particle is decremented every frame. If the strength is nonzero, then the cell the particle is in is updated with a new force dependent on the strength of the force particle. Particles are stored in a ringbuffer which is naturally sorted by age and thus strength, so overruns can be invisibly handled be overwriting the oldest living particle.

The hardware/software interface works as follows. There are four shared registers holding the current values of the old cells, the new cells, the map cells, and the framebuffer. These are swapped every frame for double buffering. A fifth register is used for synchronization. At the start of a frame, CMD_START is written to it by the software, and this tells the hardware to start. Reads from it return either STATE_BUSY or STATE_IDLE, depending

on whether or not the hardware has finished the frame. The software waits until it is idle, and starts the next frame.

Nearing the end of the semester, we ended up moving more into software. Interacting with the PLB from hardware turned out to be very difficult. We managed to get writes to DRAM from hardware working, but not reads, and did not have enough time remaining to debug our hardware against bus diagrams in Xilinx documentation. So, now our software also reads in each cell from DRAM, and farms the blending and writing out to hardware.

3.2 Hardware

We wrote hardware to handle rendering. This hardware naturally pipelines because each cell's next state is only dependent on it's neighbors' previous states and the terrain at that cell. If performance requires it we will also implement one or more of a custom framebuffer, force particle simulators, and pseudo-random number generators.

An important part of our hardware design is the cell prefetcher/cache. This is what makes it possible to access neighbors' previous states when calculating the current cell's new state. While the force interpolators are working the prefetcher will be reading in the next memory line in the next row. When a row is complete, the old version and the bottom row are both shifted up. The new version is written back to cell memory and the process begins again.

This was written as five distinct pipelined modules, plus a memory arbiter:

Fetcher - Reads cells from dram, buffers them such that interpolator can read the current cell, and the cells above and below

MapFetcher - Reads current map cell from dram, gives it to interpolator

Interpolator - Takes a column from the fetcher, and outputs a new cell whose forces are blended with its neighbors.

Renderer - Takes the map cell and the new cell and outputs a pixel, which it writes to the framebuffer in dram.

Writeback - Writes new cell to dram.

Arbiter - Muxes/Demuxes inputs and outputs of dram controller to other modules.

This also required creating a custom hardware module to do DRAM reads and writes from hardware. This means interacting with the PLB, so we created a module that let us do just this. This was very involved and required careful examination of IBM and Xilinx documentation and bus diagrams. Ultimately, we could not get reads from DRAM to work, so we instead did all of our reading from software, and let hardware do all of the sequential writes. This meant that the fetcher and the map fetcher were removed. Performance was hurt by this, but we were able to change the appearance of the game slightly and significantly reduce the amount of memory reads that had to be done, allowing it to still function.

In the end, we did not get our hardware modules to work for the public demo, and so were were forced to demonstrate the game running slowly, only in software. We did eventually get the hardware working afterwards, and the included works.

3.3 Testing

We knew testing would be a difficult part of the project, especially the software-hardware interface. We did a number of things to allow our project to be tested. We wrote a

framebuffer visualizer using SDL, so that we could simulate the hardware with a certain preloaded memory image, have it output all of its framebuffer writes to a file, and then visualize the outputs frame by frame on screen.

We also later wrote a full system simulator using the verilog programming interface, letting us test the software and hardware in tandem without resorting to resynthesizing after every small change. This let our modify-compile-test loop become much shorter, so we could make quick changes as the deadline approached.

4 Team

We had a number of issues with our team and how we structured the work. At first it was left unclear who was going to do what. Around DR1, we decided who would do what modules, though it was still hard to get the whole team assembled at the same time. We roughly divided the modules amongst ourselves, with an understanding that things would become more fleshed out as they were needed.

Initially, Karthik was to work on the parts of the board closest to hardware, doing DRAM reads and writes from hardware. The rest of us were going to do basic hardware modules in our pipeline.

However, team communication was still rather lax. After DR2, we no longer heard anything from Joon, who later dropped the course. Then, Karthik decided to drop the course the day before DR3, not having had any time to work on the coursework. This lead to a big time crunch at the end by Jeff and Rich, and ultimately, we believe, the lack of a (truly) successful public demo.

5 Lessons

There are many lessons to be taken away from our experiences in this class.

One definite one is that team dynamics are not trivial, and that clear communication is as important as anything in a successful project.

Another thing is that complex interactions between hardware and software are hard, at least given the Xilinx toolchain. Reads and writes to dram from hardware seem like something that should be able to be done by something in the Xilinx IP pack, but nothing was to be found there or online. So, doing this required careful examination of bus diagrams in IBM documentation. I would recommend groups in the future who might need this to recognize the need early and allot a good deal of time to it, because it is certainly not trivial.

I believe a decision that was particularly good was the one to use Linux on our workstations. This decision was not really recommended by the course staff, and while there were a few hiccups involving getting the toolchain installed, it made us much more productive once we started working. This is in part because our group members were all much more familiar with developing on linux than on windows. The Xilinx software worked perfectly well on windows after the initial install time.

5.1 Jeff

The largest contribution I had to the project was the work I did on the main hardware pipeline. I wrote the majority of the functional units and spent a great deal of time tracking down and fixing bugs. I am not really sure how long I spent on this, but I know that from DR3-onward, all of my time was spent in lab, and I worked on little else besides work for this class.

I thought the class was pretty good overall, but I would have liked a few more lectures, or maybe just closer lab instruction in terms of using Xilinx software and interacting with the boards. The labs didn't seem to teach much that was applicable outside of the very specific design of that lab. Learning how to use the software in general would be better than just rote clicking through menus. This might be unnecessary, as I am speaking as someone who probably has less hardware experience than most ECE majors have.

Also, video games are distracting.

6 Website

Website is located at www.contrib.andrew.cmu.edu/ johlstei/nanosiege/index.html