Code Structure of SHOOTER

Nathan Caldwell

Shooter is totally object oriented. Main is housed in shooter.cpp, so you'll want to start looking there. This doc isn't an exhaustive look at the structure, but I will first describe how the main systems are put together, then give a file by file briefing.

This is all hastily written, but it should shed some light on how the thing is put together, because it is a little overwhelming at first, I imagine.

Game Engine

For the game engine the main container object is gamespace, in gamespace.cpp The game object is timed via the neat timer code in timesensor.cpp. It's a C++ port of the timer code used in my java physics simulation. It doesn't use the normal methods for timing, it's pretty neat. So the gamespace object holds a bunch of linked lists and vectors containing the enemies, bullets, items, etc. It has a method to update everything: That is... check for collisions, update object's positions based on velocity and acceleration, and check if the player has died. For the update method, the gamespace object takes a keystate object (keystate.cpp). This holds the state of all relevent keys. This idea is VERY important for this program, since the idea of this game is to have multiple ways of letting the same game be played. If a human is playing, this thing is simply filled with the states of the actual keyboard keys. If a replay is being watched, this thing is filled with the keys the replay is telling it to press. If the AI is playing, the keystate object is filled with the keys the AI wants to press.

Graphics

The graphics use Irrlicht sprite routines. In shooter.cpp, I make a call to a method of gamespace that looks at all of its objects, and fills a spritenote (spritenote.cpp) object with the x, y, and sprite # to draw. Then shooter.cpp takes this list and uses Irrlicht to draw each 'note' in the screen. For stars, Gamespace keeps track of all the x, y and z. Each cycle I move them a little down the screen (speed based of course on their z value), and then draw a pixel there. I scale the colour based on the z value as well, so starts that are further away are darker.

Main AI

The player AI is housed in aiplayer.cpp. The AI works totally by FSM and motion blending. The FSM has 3 states: panic, item, and normal. Normal is the default that is used in any situation. Item is used whenever there is an item on the screen to pick up. Panic is used whenever a bullet comes close to the ship. The motion blending pushes the ship around, and its variables are based on the state. This is described in bloody mathematical detail in the paper about the AI.

Enemies

Enemies run on FSMs, but these machines are generally used simply to script the actions of the enemies. I can easily make simple enemies, or even complex multi phase boss characters using the code I have built. I would eventually like to use lua script to run these guys. Basically, there is a base class in enemybase.cpp, and the waypoint base (the states in the FSM) also in enemybase.cpp. In myenemies.cpp I actually derive the enemies I use from the base class. To see how I get the complex enemy scripted motion, look in myenemies.

Bullets and Barrages
The bullets and barrages are housed in bullet.cpp and barrage.cpp. A bullet has physical properties including linear motion, radial motion, and rotational motion. A barrage is a collection of bullets held in a vector, with some utility methods to set up some complex barrages such as the ones you see in my demo waves. To use a barrage, you call the getbarrage method, and feed it a linked list of bullets. It adds the barrage bullets to the linked list, thus adding the complex pattern to the game world list of bullets.

Genetic Algorithm
I came up with an interesting way to do the genetic algorithm. The AI takes a bunch of variables, which are stored in a dna object (dna.cpp). A generation object (generation.cpp) holds 30 of these. When the genetic algorithm is run, the program simulates all 30 bots (without graphics and timer code, so it pumps as fast as the cpu will let it) and the fitness is their final score. Next I choose a certain number of them at random, and from this selection choose the best one and add it to the mating pool. I repeat until there are 5 dna objects in the mating pool. After that, 2 are chosen at random from the mating pool, and mated. this is done 10 times, and the results stored in a new generation object. Next, 10 more are created from mating, but these ones are mutated. Finally, 10 more are created and mutated severely. This accounts for all 30 bots in the next generation. The process is then repeated. It works fairly well if good number are entered. Check out fromcryo2.dna to see a pretty decent one made from evolution.

Replay Saving
Replay saving is all handled in shooter.cpp. When making a replay, the user's key information is logged in a vector as he plays. The keys aren't just dumped in every cycle, however. The keys are scanned every cycle for changes from last cycle's keystates. If no change is seen, nothing is logged. If a key differs from last cycle, the cycle number , key ID, and key state are logged (I could reduce the size of replay files by 1/3 by giving the ax to key state logging, I just realized, since the key states are boolean values and only a flip is needed... but I don't have time to mess with it right now). When watching replays, it simply has a keystate that only gets changed when the cycle number matches that of the next key change note in the replay file. There is a sample replay include, goodrun.rep, so you can check out that (or of course just make your own).

Play Theory
I put a bit of thought into how the game should be played. I wanted complex bullet patterns... but for the amazing number of bullet on the screen, the player's hit box needed to be small. I could just make the player small, but that would kind of suck. I wanted a decently sized ship for the player to control, so I just made the very center of him the hit location. To show where you can be hit, I drew a little window on the ship that is approximately the hit location. To get an accurate hit location display, hold shift. This is known as focus mode. Not only does it allow the hit location to be seen, but it also allows the player ship to move more slowly, for better dodging. (The AI does, in fact, use this feature!). For scoring, each hit you get on an enemy nets you 10 points. Each kill gets you a specific number of points for that enemy. Each score item (the blue ones) you pick up give you a number of points proportional to you height up the screen. This encourages players not not dwell at the screen bottom where the bullet density is less. Same for the AI. It's actually effective for getting genetically created AI to stay away from the bottom of the screen, too! Not just human players...
Finally for player movement, I tried at first only having the player move when pressing a direction key (perfectly synched with pressing), but it felt strange like there was no weight to the player. I opted to treat the player as a particle with a super high friction value on his movement. It's not conciously noticeable that he skids a pixel or two after you let off before he stops, but it just feels so much better. One of those things where it feels nice but you couldn't quite place your finger on why. I think that it's

important to have this sort of quality in player control.

File by file description in no particular order:
Shooter.cpp – houses main, initializes the graphics, replays, gamespace, etc. Contains the main game loop. Holds the cheap “main menu” stuff.

Gamespace.h / Gamespace.cpp – Main game code. Holds player object, items, bullets, enemies, etc. Also takes care of motion integration, and reporting of sprite locations. Also takes care of the spiffy particle effects.

Dna.h / Dna.cpp – Holds the big fat wad of variables used by the AI. Also contains methods to mate and mutate the things.

AIPlayer.h / AIPlayer.cpp – Holds the code for the ai player. This thing is more thoroughly described in the other paper included. Basically just takes a dna object and works its magic.

Barrage.h / Barrage.cpp – Holds the code for building and storing barrages (the groups of bullets that I shoot off)

Bullet.h / Bullet.cpp – Holds the code for bullets.

Enemybase.h / Enemybase.cpp – Holds the base object for enemies and enemy waypoints.

Irrevent.h / Irrevent.cpp – talks to the irrlicht input functions (so I don't have to piddle around with windows programming)

Item.h / Item.cpp – Holds the base class for items.

Keystate.h / Keystate.cpp – holds the key information fed to the gamespace object.

Linear.h / Linear.cpp – Linear algebra library I never ended up putting to use.

Myenemies.h / Myenemies.cpp – The code for the specific enemies I used to make the demo levels.

Player.h / Player.cpp – The object holding the player info.

PointItem.h / PointItem.cpp – Holds the code for the point item.

Random.h / Random.cpp – a very nice random library.

Scoreitem.h / Scoreitem.cpp – Holds the code for the score item.

SpriteNote.h / SpriteNote.cpp – The object the shuttle sprite information from the game to the rendering engine.

Star.h / Star.cpp – A dumb little object holding the background parallaxed star info.

TimeSensor.h / Timesensor.cpp – The neat timer code

TwoVector.h / TwoVector.cpp – a quick and dirty 2 dimensional vector object.